



HAL
open science

On the Optimization of Iterative Programming with Distributed Data Collections

Sarah Chlyah, Nils Gesbert, Pierre Genevès, Nabil Layaïda

► **To cite this version:**

Sarah Chlyah, Nils Gesbert, Pierre Genevès, Nabil Layaïda. On the Optimization of Iterative Programming with Distributed Data Collections. 2021. hal-02066649v5

HAL Id: hal-02066649

<https://inria.hal.science/hal-02066649v5>

Preprint submitted on 2 Mar 2021 (v5), last revised 24 May 2022 (v6)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On the Optimization of Iterative Programming with Distributed Data Collections

SARAH CHLYAH, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

NILS GESBERT, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

PIERRE GENEVÈS, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

NABIL LAYAÏDA, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

Big data programming frameworks are becoming increasingly important for the development of applications for which performance and scalability are critical. In those complex frameworks, optimizing code by hand is hard and time-consuming, making automated optimization particularly necessary. In order to automate optimization, a prerequisite is to find suitable abstractions to represent programs; for instance, algebras based on monads or monoids to represent distributed data collections. Currently, however, such algebras do not represent recursive programs in a way which allows for analyzing or rewriting them. In this paper, we extend a monoid algebra with a fixpoint operator for representing recursion as a first class citizen and show how it enables new optimizations. Experiments with the Spark platform illustrate performance gains brought by these systematic optimizations.

1 INTRODUCTION

Ideas from functional programming play a major role in the construction of big data analytics applications. For instance they directly inspired Google's Map/Reduce [7]. Big data frameworks (such as Spark [25] and Flink [5]) further built on these ideas and became prevalent platforms for the development of large-scale data intensive applications. The core idea of these frameworks is to provide intuitive functional programming primitives for processing immutable distributed collections of data.

Writing efficient applications with these frameworks is nevertheless not trivial. Let us consider for instance the problem of finding the shortest paths in a large scale graph. We could write the Spark/Scala program in Fig 1 to solve it. The `shortestPaths()` function takes as input a graph `R` of weighted edges (`src`, `dst`, `weight`) and returns the shortest paths between each pair of nodes in the graph. The loop (in lines 6 to 14 of Fig 1) computes all the paths in the graph and their lengths; to get new paths, edges from the graph get appended to the paths found in the previous iteration using the join operation. Then `reduceByKey` operation is used to keep the shortest paths.

```

1  def shortestPaths(R:RDD[(Int,Int,Int)]) = {
2    var ret = R
3    var X: RDD[(Int, Int, Int)] = R
4    var new_cnt = ret.count()
5    var cnt = new_cnt
6    do {
7      cnt = new_cnt
8      X = X.map({case (x,y,l1) => (y,(x,l1)) })
9          .join(R.map({ case (z,t,l2) => (z,(t,l2)) })))
10         .map({case (_,((x,l1),(t,l2))) => (x,t,l1+l2) })
11         .subtract(ret)
12         ret = ret.union(X).distinct()
13         new_cnt = ret.count()
14     } while (new_cnt > cnt)
15     ret.map({case (x,y,l) => ((x,y),l)}).reduceByKey(min)
16 }

```

Fig. 1. Shortest paths program.

Spark performs the join and distinct operations by transferring the datasets (arguments of the operations) across the workers so as to ensure that records having the same key are in the same partition for join, and that no record is repeated across the cluster for distinct. Hence, for optimizing such programs, the programmer needs to take this data exchange into account as well as other factors like the amount of data processed by each worker and its memory capacity, the network overhead incurred by shuffles, etc. One optimization that can be done to reduce data exchange in this program is to assign each worker a part of the graph and make it compute the paths in the graph that start from its own part. This optimization leads to the following program (Fig 2.) which is not straightforward to write, less readable, and requires the programmer to give his own local version of dataset operators (such as join) that are going to be used to perform the local computations on each worker.

Another possible optimization is to put the reduceByKey operation inside the loop to keep only the shortest paths at each iteration because each subpath of a shortest path is necessarily a shortest path. More generally, finding such program rewritings can be hard. First, it requires guessing which program parts affect performance the most and could potentially be rewritten more efficiently. Second, assessing that the rewriting performs better can hardly be determined without experiments. During such experiments, the programmer might rewrite the program possibly several times, because he has limited clues of which combination of rewritings actually improves performance.

In this paper, we explore the foundations for the automatic transformation and optimization of Spark programs. Algebraic foundations in particular are an active research topic [2, 8]. The purpose of the algebraic formalism is to represent a program in terms of algebraic operators that can be analysed and transformed so as to produce a program that executes faster. Transformation-based optimizations are done through rewrite rules that transform an algebraic expression to an equivalent, yet more efficient, one. In the context of big data applications, considered algebras must be able to capture distributed programs on big data platforms and provide the appropriate primitives to allow for their optimization. One example of optimizations is to push computations as close as possible to where data reside.

When programming with big data frameworks, data is usually split into partitions and both data partitions and computations are distributed to several machines. These partitions are processed in

```

1  def shortestPaths(R:RDD[(Int,Int,Int)]) = {
2    val dictR = LocalOps.to_dict(((x:(Int,Int,Int)) => x._1),
3      (x:(Int,Int,Int)) => x, sc.broadcast(R.collect()).value)
4    var r = R.mapPartitions(part => {
5      var ret = part.toList
6      var X = ret
7      var cnt = ret.size
8      var new_cnt = cnt
9      do {
10       count = new_count
11       X = LocalOps.join(LocalOps.to_dict(((x:(Int,Int,Int)) => x._2),
12         (x:(Int,Int,Int)) => x, X), dictR)
13         .map({case (k, ((x,y,l), (a,b,m))) => (x,b,l+m)}) diff ret
14       ret = (ret ++ X).distinct
15       new_count = ret.size
16     } while (new_cnt > cnt)
17     ret.toIterator
18   })
19   r.distinct().map({case (x,y,l) => ((x,y),l)}).reduceByKey(min)
20 }

```

Fig. 2. Shortest paths program with less data exchange.

parallel and intermediate results coming from different machines are combined, so that a unique final result is obtained, regardless of how data was split initially. This imposes a few constraints on computations that combine intermediate results. Typically, functions used as aggregators must be associative. For this reason, we consider that the monoid algebra is a suitable algebraic foundation for taking this constraint into account at its core. It provides operations that are monoid homomorphisms, which means that they can be broken down to the application of an associative operator. This associativity implies that parts of the computation can actually be performed in parallel and combined to get the final result.

A significant class of big data programs are iterative or recursive in nature (PageRank, k-means, shortest-path, reachability, etc.). Iterations and recursions can be implemented with loops. Depending on the nature of the computations performed inside a loop, the loop might be evaluated in a distributed manner or not. Furthermore, certain loops that can be distributed might be evaluated in several ways (global loop on the driver¹, parallel loops on the workers, or a nested combination of the latter). The way loops are evaluated in a distributed setting often has a great impact on the overall program execution cost. Obviously, the task of identifying which loops of an entire program can be reorganized into more efficient distributed variants is challenging. This often constitutes a major obstacle for automatic program optimization. In the algebraic formalism, having a recursion operator makes it possible to express recursion while abstracting away from how it is executed. The execution plan is then decided after analysing the program.

The goal of this work is to introduce a gain in automation of distributed program transformation towards more efficient variants. We focus especially on recursive programs (that compute a fixpoint). For this purpose, we propose an algebra capable of capturing the basic operations of distributed computations that occur in big data frameworks, and that makes it possible to express rewriting rules that rearrange the basic operations so as to optimize the program. We build on the monoid

¹In Big Data frameworks such as Spark, the *driver* is the process that creates tasks and sends them to be executed in parallel by *worker* nodes

algebra introduced in [8, 10] that we extended with an operator for expressing recursion. This monoid algebra is able to model a subset of a programming language \mathcal{L} (for instance Scala), that expresses computations on distributed platforms (for instance Spark).

Contributions. Our contributions are the following:

- (1) An extension of the monoid algebra with a fixpoint operator. This enables the expression of recursion in a more functional way than an imperative loop and makes it possible to define new rewriting rules;
- (2) New optimization rules for terms using this fixpoint operator:
 - We show that under reasonable conditions, this fixpoint can be considered as a monoid homomorphism, and can thus be evaluated by parallel loops with one final merge rather than by a global loop requiring network overhead after each iteration;
 - We also present new rewriting rules with criteria to push filters through a recursive term, for filtering inside a fixpoint before a join, and for pushing aggregations into recursive terms;
 - Finally, we present experimental evidence that these new rules generate significantly more efficient programs.

2 THE μ -MONOIDS ALGEBRA

In this section, we describe a core calculus, which we call μ -monoids, intended to model a subset of a programming language \mathcal{L} (e.g. Scala²) that is used for computations on a big data framework (through an API provided by the framework). μ -monoids aims at being as general as possible, while focusing on formalizing computations subject to optimization. Dataset manipulations are captured as algebraic operations, and specific operations on elements of those datasets are captured as functional expressions that are passed as arguments to some of the algebraic operations. In μ -monoids, we formalize *some* of those functional constructs, specifically the ones that we need to analyse in the algebraic expressions. For example, some optimization rules need to analyse the pattern and body of flatmap expressions in order to check whether the optimization can take place.

Making explicit only the shapes that are interesting for the analysis enables to abstract from the specific programming language \mathcal{L} that we optimize. This way, constructs of \mathcal{L} other than those which we model explicitly are represented as *constants* c , as they are going to be left to \mathcal{L} 's compiler to typecheck and evaluate. We only assume that every constant c has a type $type(c)$ which is either a basic type or a function type, and that, when its type is $t_1 \rightarrow t_2$, it can be applied to any argument of type t_1 to yield results of type t_2 .

We first describe the data model we consider, then in Sec. 2.2 we introduce the syntax of our core calculus. We then proceed to give a denotational semantics for our specific constructs in Sec. 2.3 and discuss evaluation of expressions in Sec. 2.5.

2.1 Data model: distributed collections of data

In big data frameworks, a data collection is divided into sub-collections stored into each machine. A collection can be in the form of a bag (a structure where the order is not important and where elements can be repeated), a list (a sequenced bag), or a set (a bag where elements do not repeat). In the context of this paper and for the sake of simplicity, we will focus on bags, although the operations we present can easily be defined for lists. Sets with no duplicates are impractical to

²Major Bigdata frameworks like Spark and Flink provide a Scala API and is implemented in Scala which makes Scala a suitable language for our work. Scala also provides reflection which allows generic Scala constructs to be part of the algebra as we will explain later.

implement in a distributed context but we can assume that the language \mathcal{L} provides a distinct operation which removes all duplicates from a bag.

In order to enable algebraic datatypes, we assume an infinite set of *constructors* C which can be applied to any number of values. We assume this set contains the special constructors `True`, `False` and `Tuple` for which we will define some syntactic sugar.

The syntax of considered data values is defined as follows:

| | | |
|---------|--|----------------------|
| $v ::=$ | c | constant |
| | $ C(v_1, v_2, \dots, v_n)$ | n -ary constructor |
| | $ \{v_1\} \uplus \{v_2\} \uplus \dots \uplus \{v_n\}$ | bag |

in which a bag is seen as the union of its singletons where \uplus denotes the bag union operator. As mentioned previously (Sec. 2), a constant c can be any value from the language \mathcal{L} (in particular any function) that is not explicitly defined in our syntax.

We define the following syntax for types:

| | | | | | |
|-----------|---|----------------|---------|-----------------------|----------------------|
| $t_l ::=$ | \mathbb{B} | local type | $t ::=$ | type | |
| | $ C_1[t_1, \dots, t_l] \parallel \dots \parallel C_n[t_1, \dots, t_l]$ | basic type | | $ t_l$ | |
| | $ \text{Bag}_l[t_l]$ | sum type | | $ \text{Bag}_d[t_l]$ | distributed bag type |
| | | local bag type | | $ t \rightarrow t$ | function type |

where \mathbb{B} represents any arbitrary basic type (i.e., considered as a constant atomic type in our formalism).

We also define product types $t_1 \times \dots \times t_n$ as syntactic sugar for `Tuple`[t_1, \dots, t_n].

In sum types, all constructors have to be different and their order is irrelevant.

For a given type t , we denote by $\text{Bag}_l[t]$ the type of a local bag and by $\text{Bag}_d[t]$ the type of a distributed bag of values of type t . Notice that we can have distributed bags of any data type t including local bags, which allows us to have nested collections. We allow data distribution only at the top level though (distributed bags cannot be nested).

Some operators over bags can be defined similarly, regardless of whether bags are local or distributed. For convenience, we thus denote the type of bags, either distributed or not as: $\text{Bag}[t] ::= \text{Bag}_l[t] \mid \text{Bag}_d[t]$. For example, the bag union operator $\uplus : \text{Bag}[t] \times \text{Bag}[t] \rightarrow \text{Bag}[t]$ is defined for both local and distributed bags. We consider that whenever any argument of \uplus is a distributed bag then the result is a distributed bag as well; if, on the contrary, both arguments are local bags then the result may be either a local bag or a distributed bag. We usually do not need to distinguish the cases, but when it is relevant to do so (as in Sec. 2.5.2), we use a vertical bar $|$ to indicate nonlocal union of local bags into a distributed one.

2.2 The μ -monoids syntax

μ -monoids syntax contains mainly primitives for processing distributed data collections. It is built on the monoid algebra proposed by Fegaras [8] and extended with a fixpoint operator for expressing recursion. Expressions consist of functional expressions and algebraic operations (flatmap, group by key...) performed on collections. Functional expressions can appear as arguments of those algebraic operations. For example $\text{fmap}(\lambda (a \rightarrow \{a + 1\}), b)$ has two arguments: a λ -expression $\lambda (a \rightarrow \{a + 1\})$ (a function that returns a singleton of the incremented value of its argument), and a second argument b which is a variable (referencing some collection).

The syntax of expressions is formally defined as follows:

| | | | |
|-------|-----|---|---|
| π | ::= | $a \mid C(\pi_1, \pi_2, \dots, \pi_n)$ | pattern: variable, constructor pattern |
| e | ::= | $c \mid a \mid \{e\}$ | expression: constant, variable, singleton |
| | | $\lambda (\pi_1 \rightarrow e_1 \mid \dots \mid \pi_n \rightarrow e_n)$ | function with pattern matching |
| | | $e \ e \mid C(e_1, e_2, \dots, e_n)$ | application, constructor expression |
| | | $\text{flatMap}(e, e) \mid \text{reduce}(e, e) \mid \text{groupBy}(e)$ | flatMap, reduce, group by key |
| | | $\text{reduceByKey}(e, e) \mid \text{cogroup}(e, e) \mid \text{join}(e, e)$ | reduce by key, cogroup, join by key |
| | | $\mu (e, e)$ | fixpoint |

To this, we add the following as syntactic sugar:

- (e_1, \dots, e_n) with no constructor is an abbreviation for: $\text{Tuple}(e_1, \dots, e_n)$
- if e then e_1 else e_2 is an abbreviation for: $\lambda (\text{True} \rightarrow e_1 \mid \text{False} \rightarrow e_2) \ e$
- Constants c can also represent functions (defined in the language \mathcal{L}). We consider operators such as the bag union operator \uplus as constant functions of two arguments and use the infix notation as syntactic sugar.

Example:

$$\mu(C, \lambda (X \rightarrow \text{flatMap}(\lambda (x \rightarrow \text{flatMap}(\lambda (c \rightarrow \text{if contains } x \ c \text{ then } \{\} \text{ else } \{x + c\}), C)), X)))$$

This expression computes the set of all possible words (with no repeated letters) that can be formed from a set of characters C . The expression in bold represents a function (we call it `appendToWords`) that returns a new set of words from a given set of words X by appending to each of the words in X each letter in C whenever possible. `contains` is a function defined in \mathcal{L} , it checks whether the first argument is contained in the second argument.

The fixpoint operator computes the following, where we denote X_i the result at the iteration i and consider $C = \{a, b, c\}$ – the fixpoint is reached in 3 steps:

$$\begin{aligned} X_0 &= C \\ X_1 &= \text{appendToWords}(C) \cup C = \{ab, ac, ba, bc, ca, cb, a, b, c\} \\ X_2 &= \text{appendToWords}(X_1) \cup X_1 = \{abc, acb, bac, bca, cab, cba, ab, ac, ba, bc, ca, a, b, c\} \\ X_3 &= \text{appendToWords}(X_2) \cup X_2 = \{abc, acb, bac, bca, cab, cba, ab, ac, ba, bc, ca, a, b, c\} \end{aligned}$$

Well-typed terms. In order to exclude meaningless terms, we can define typing rules for algebraic terms. These rules are quite standard; we give them for reference in Appendix A.

2.3 μ -monoids denotational semantics

2.3.1 Monoid homomorphisms. The semantics of the μ -monoids algebra extends the semantics of the monoid algebra [8]. We first recall basic definitions from the monoid algebra and then present the fixpoint operation that we introduce.

Briefly, a *monoid* is an algebraic structure (S, \oplus, e) where S is a set (called the carrier set of the monoid), \oplus an associative binary operator between elements of S , and e is an identity element for \oplus . A *monoid homomorphism* h from (S, \oplus, e) to (S', \otimes, e') is a function $h : S \rightarrow S'$ such that $h(x \oplus y) = h(x) \otimes h(y)$ and $h(e) = e'$.

Given any type α , we can consider the set of lists (finite sequences) of elements of α ; if we equip this set with the concatenation operator $++$ (list union), it yields a monoid $(\text{List}[\alpha], ++, [])$ (algebraically called the free monoid on α), whose identity element is the empty list. Let $U_{\text{List}} : \alpha \rightarrow \text{List}[\alpha]$ be the singleton construction function. This monoid has the following universal property: let (S, \otimes, e) be any monoid and $f : \alpha \rightarrow S$ any function, then there exists exactly one monoid homomorphism, denoted H_f^\otimes , such that $H_f^\otimes : \text{List}[\alpha] \rightarrow S$ and $H_f^\otimes \circ U_{\text{List}} = f$. This homomorphism

can be simply defined by:

$$H_f^\otimes([a_1, \dots, a_n]) \stackrel{\text{def}}{=} f(a_1) \otimes \dots \otimes f(a_n)$$

For example, given the monoid $(\text{Int}, +, 0)$ and the function $\text{one} : x \rightarrow 1$, we have that H_{one}^+ is the monoid homomorphism which counts the elements of its input list.

Lists are a type of collections where the order of elements is important. Other types of collections are monoids as well and can be defined from the list monoid as follows. Let us consider a set of algebraic laws for a binary operator, for example commutativity ($a \otimes b = b \otimes a$) or the ‘graphic identity’ $a \otimes b \otimes a = a \otimes b$. It is possible to define the quotient of the list monoid by a set of such laws. For example, the congruence induced by commutativity relates all lists which contain exactly the same elements in different orders; i. e. the quotient of the monoid $\text{List}[\alpha]$ by commutativity is isomorphic to $(\text{Bag}[\alpha], \uplus, \{\})$, the bag monoid, where \uplus is the bag union operator and $\{\}$ the empty bag. Such quotients of the free monoid have been termed *collection monoids* by Fegaras et al. Another notable example of collection monoid is the monoid of finite sets on α , $(\text{Set}[\alpha], \cup, \emptyset)$, obtained by quotienting with both commutativity and the graphic identity³.

Collection monoids inherit the universal property of lists in the following way: let $(T[\alpha], \oplus, e_T)$ be a collection monoid noted (\oplus) and (β, \otimes, e) a monoid noted (\otimes) , where α and β are arbitrary types. Suppose \otimes obeys all the algebraic laws of \oplus , then for any function $f : \alpha \rightarrow \beta$, there exists a unique homomorphism⁴ $H_f^\otimes : T[\alpha] \rightarrow \beta$ such that $H_f^\otimes \circ U_T = f$.

As a slight abuse of notation, we will sometimes refer to ‘the monoid \uplus ’ for example to designate the monoid of bags on an unspecified type α .

As mentioned previously in Section 2.1, we will focus on bag monoids (local and distributed bags) as a start monoid for the homomorphic operations. Those operations share the same denotational semantics for local and distributed bags (they return the same values regardless of whether those values are distributed or not).

Summary. To summarise the earlier definitions in the case of bags, if (β, \otimes, e) is a commutative monoid and $f : \alpha \rightarrow \beta$ is any function, the monoid homomorphism H_f^\otimes from the collection monoid $(\text{Bag}[\alpha], \uplus, \{\})$ to the monoid (β, \otimes, e) (denoted $\uplus \rightarrow \beta$) satisfies the following:

$$\begin{aligned} H_f^\otimes(X \uplus Y) &= H_f^\otimes(X) \otimes H_f^\otimes(Y) \\ H_f^\otimes(\{x\}) &= f(x) \\ H_f^\otimes(\{\}) &= e \end{aligned}$$

Here we can see that the associativity property of \uplus and \otimes is interesting in the context of distributed programming because it is possible to compute $H_f^\otimes(X)$ by dividing X into multiple parts, applying the computation on each part independently, then gathering the results using the \otimes operator without leading to erroneous results.

2.3.2 Restrictions on the μ -monoids operations. In order to be well defined, some operations need to fulfill certain criteria as shown below

- $\text{reduce}(f, A)$ and $\text{reduceByKey}(f, A)$: f is associative and commutative
- $\mu(R, \varphi)$: φ is a monoid homomorphism $\uplus \rightarrow \uplus$

³As a less notable example, the graphic identity alone yields the monoid $\text{OrderedSet}[\alpha]$.

⁴It may be worth mentioning that, thanks to this property, while a collection monoid $T[\alpha]$ has the structure of a monoid, the constructor T itself also defines a *monad*, whose unit function is U_T (singleton construction) and whose monadic operations map and flatMap can be defined from the universal property.

$$\begin{aligned}
\text{flmap}(f, A) &= \bigoplus_{a \in A} f(a) & \text{reduce}(f, A) &= r_N, \text{ where } A = \{a_1, a_2, \dots, a_N\}, r_n = f(r_{n-1}, a_n), r_1 = a_1 \\
\text{groupby}(A) &= \{(k, \{v \mid (k, v) \in A\}) \mid k \in \text{keys}(A)\} \\
\text{reduceByKey}(f, A) &= \{(k, \text{reduce}(f, \{v \mid (k, v) \in A\})) \mid k \in \text{keys}(A)\} \\
\text{cogroup}(A, B) &= \{(k, (\{v \mid (k, v) \in A\}, \{w \mid (k, w) \in B\})) \mid k \in \text{keys}(A) \cup \text{keys}(B)\} \\
\text{join}(A, B) &= \{(k, (v, w)) \mid (k, v) \in A \wedge (k, w) \in B\} & \mu(R, \varphi) &= \bigcup_{n \in \mathbb{N}} \Phi^{(n)}(R), \text{ where } \Phi: X \mapsto X \cup \varphi(X)
\end{aligned}$$

where: the comprehensions denote bag comprehensions; $\text{keys}(A) = \text{distinct}(\{(k \mid (k, a) \in A)\})$; and \cup is distinct union of bags.

Fig. 3. Denotational semantics

The user needs to provide terms that satisfy these criteria since they cannot be verified statically. However, for the second criteria we can identify a subset of homomorphisms $\mathfrak{U} \rightarrow \mathfrak{U}$ that can be statically checked. It is the set of terms φ of the form $\lambda (X \rightarrow T(X))$ where $T(X)$ is defined as follows⁵:

$$\begin{aligned}
T(X) &::= \\
& \quad X \\
& \quad | \text{flmap}(f, T(X)) && X \text{ does not appear in } f \\
& \quad | \text{join}(T(X), A) && X \text{ does not appear in } A \\
& \quad | \text{join}(A, T(X)) && X \text{ does not appear in } A
\end{aligned}$$

Figure 3 gives the denotational semantics of the main algebraic operations. Note that this set of operations is not minimal: some operations can be defined in terms of others, for example `reduceByKey` can be obtained by combining `reduce` and `groupby`. However we prefer to include them all in the main syntax for clarity.

2.3.3 μ -monoids operators as monoid homomorphisms. We first give a small description of the monoid operations, then we define the fixpoint operator. In Figure 4, we explain how these operations (except the fixpoint) are monoid homomorphisms which can be defined as H_f^\otimes for appropriate f and \otimes .

The flatmap operator. We consider a function $f : \alpha \rightarrow \text{Bag}[\beta]$. `flmap`(f, X) applies f on each element in the bag X and returns a dataset that is the union of all results.

The reduce operator. `reduce`(\oplus, X) reduces the elements of the input dataset by combining them with the \oplus operator. For example: `reduce`($+$, $\{1, 4, 6\}$) = 11.

The groupby operator. `groupby`(A) takes a bag of elements in the form (k, v) , where k is considered the *key* and v the *value*, and returns a bag of elements in the form (k, V) where V is the bag of all elements having the same key in the input dataset. Thus, each key appears exactly once in the result. For example, `groupby`($\{(1, 2), (1, 4), (2, 2), (2, 1), (1, 3)\}$) = $\{(1, \{2, 4, 3\}), (2, \{2, 1\})\}$.

The reduceByKey operator. `reduceByKey`(\oplus, X) takes as argument a bag of elements in the form (k, v) and combines all values v having the same key k into a single one using the \oplus operator. For example: `reduceByKey`($+$, $\{(1, 2), (1, 4), (2, 2), (2, 1), (1, 3)\}$) = $\{(1, 9), (2, 3)\}$.

⁵This set corresponds to the composition of homomorphisms that are known to be $\mathfrak{U} \rightarrow \mathfrak{U}$ because the composition of homomorphisms $\mathfrak{U} \rightarrow \mathfrak{U}$ is also a homomorphism $\mathfrak{U} \rightarrow \mathfrak{U}$.

The cogroup operator. $\text{cogroup}(A, B)$ takes two collections of elements of the form (k, v) and (k, w) and returns a collection of elements of the form $(k, (V, W))$ where V and W are the sets of v values and w values having the same key k .

The join operator. $\text{join}(A, B)$ takes two collections of elements of the form (k, v) and (k, w) , and returns a collection of elements of the form $(k, (v, w))$, one for each pair (v, w) of values having the same key k . If a key appears n times in one input dataset and m times in the other, it appears nm times in the result.

The Fixpoint operator. Let R be a bag and φ a lambda expression. $\mu(R, \varphi)$ is defined as the smallest fixpoint of the function $\Psi: X \rightarrow X \cup \psi(X)$ ⁶, where $\psi(X) = R \uplus \varphi(X)$. R is called the constant part of the fixpoint and φ the variable part⁷.

It can be shown (see Appendix B.1) that when φ is a monoid homomorphism $\uplus \rightarrow \uplus$, the fixpoint exists (Ψ has a fixpoint) and can be reached from the successive application of φ starting from R (as shown in Figure 3). We thus restrict our language to this particular kind of fixpoint.

Under this criteria, the μ operator is a monoid homomorphism $H_f^U: \uplus \rightarrow \cup$, where $f(a) = \mu(\{a\}, \varphi)$:

$$\mu(R_1 \uplus R_2, \varphi) = \mu(R_1, \varphi) \cup \mu(R_2, \varphi)$$

2.4 Examples

We present in this section examples of recursive programs expressed in μ -monoids.

Transitive closure (TC).

$$\mu(R, \lambda (X \rightarrow \text{flmap}(\lambda ((b, (a, c)) \rightarrow \{(a, c)\}), \text{join}(\text{flmap}(\lambda ((a, b) \rightarrow \{(b, a)\}), X), R))))$$

where R is a dataset of tuples (source, destination) representing the edges of a graph.

This expressions computes the entire transitive closure of the input graph R .

The sub-expression $\text{join}(\text{flmap}(\lambda ((a, b) \rightarrow \{(b, a)\}), X), R)$ joins a path from X with a path from R when the target node of the first path corresponds to the start node of the second path. So, at each iteration, the paths in X obtained in the last iteration get appended with edges from R whenever possible. The computation ends when no new paths are found.

Shortest path (SP).

$$\begin{aligned} &\text{reduceByKey}(\text{min}, \\ &\mu(R, \lambda (X \rightarrow \text{flmap}(\lambda ((b, ((a, l_1), (c, l_2))) \rightarrow \{(a, c, l_1 + l_2)\}), \\ &\text{join}(\text{flmap}(\lambda (((a, b), l_1) \rightarrow \{(b, (a, l_1))\}), X), \text{flmap}(\lambda ((b, c, l_2) \rightarrow \{(b, (c, l_2))\}), R)))))) \end{aligned}$$

where R is a dataset of tuples (source, destination, weight) representing the weighted edges of a graph.

The expression computes the shortest path between each pair of nodes in the input graph R . New paths are computed by performing a transitive closure while summing the lengths of the joined paths. Finally, the `reduceByKey` operation keeps the shortest paths between each pair of nodes.

⁶In the context of bags where $\psi: \text{Bag}[\alpha] \rightarrow \text{Bag}[\beta]$, \cup corresponds to a bag union with no duplicates.

⁷We have $\psi(\{\}) = R$ because $\varphi(\{\}) = \{\}$ since φ is a homomorphism $\uplus \rightarrow \uplus$.

$$\begin{aligned}
\text{fimap}(f, \cdot) &= H_f^\uplus : (\text{Bag}[\alpha], \uplus, \{\}) \rightarrow (\text{Bag}[\beta], \uplus, \{\}) \quad (\text{with } f : \alpha \rightarrow \text{Bag}[\beta]) \\
\text{reduce}(\oplus, \cdot) &= H_{id}^\oplus : (\text{Bag}[\alpha], \uplus, \{\}) \rightarrow (\beta, \oplus, e_\oplus) \\
\text{groupby}(\cdot) &= H_f^\uparrow : (\text{Bag}[\alpha \times \beta], \uplus, \{\}) \rightarrow (\text{Bag}[\alpha \times \text{Bag}[\beta]], \uparrow, \{\}) \\
\text{where } f &: \begin{array}{l} \alpha \times \beta \rightarrow \text{Bag}[\alpha \times \text{Bag}[\beta]] \\ (k, v) \mapsto \{(k, \{v\})\} \end{array} \\
\text{and } \{(k, b_1)\} \uparrow \{(k', b_2)\} &= \begin{cases} \{(k, b_1 \uplus b_2)\} & \text{if } k = k' \\ \{(k, b_1), (k', b_2)\} & \text{otherwise} \end{cases} \\
\text{reduceByKey}(\oplus, \cdot) &= H_{U_{\text{Bag}}}^\uparrow : (\text{Bag}[\alpha \times \beta], \cup, \{\}) \rightarrow (\text{Bag}[\alpha \times \beta], \uparrow_\oplus, \{\}) \\
\text{where } \{(k, b_1)\} \uparrow_\oplus \{(k', b_2)\} &= \begin{cases} \{(k, b_1 \oplus b_2)\} & \text{if } k = k' \\ \{(k, b_1), (k', b_2)\} & \text{otherwise} \end{cases} \\
\text{join}(A, \cdot) &= H_f^\uplus : ((\text{Bag}[\alpha \times \gamma], \uplus, \{\}) \rightarrow (\text{Bag}[\alpha \times (\beta \times \gamma)], \uplus, \{\}) \quad (\text{with } A : \text{Bag}[\alpha \times \beta]) \\
\text{where } f &: \begin{array}{l} \alpha \times \gamma \rightarrow \text{Bag}[\alpha \times (\beta \times \gamma)] \\ (k, v) \mapsto \{(k, (w, v)) \mid (k, w) \in A\} \end{array}
\end{aligned}$$

Similarly for $\text{join}(\cdot, A)$.

$$\begin{aligned}
\text{cogroup}(\cdot, \cdot) &= H_{f_1, f_2}^\uparrow : ((\text{Bag}[\alpha \times \beta], \uplus, \{\}) \times (\text{Bag}[\alpha \times \gamma], \uplus, \{\})) \rightarrow \text{Bag}[\alpha \times (\text{Bag}[\beta]) \times \text{Bag}[\gamma]] \\
\text{where :} &
\end{aligned}$$

$$\begin{aligned}
f_1 : \alpha \times \beta &\rightarrow \text{Bag}[\alpha \times (\text{Bag}[\beta]) \times \text{Bag}[\gamma]] \\
(k, v) &\mapsto \{(k, (\{v\}, \{\}))\} \\
f_2 : \alpha \times \gamma &\rightarrow \text{Bag}[\alpha \times (\text{Bag}[\beta]) \times \text{Bag}[\gamma]] \\
(k, v) &\mapsto \{(k, (\{\}, \{v\}))\}
\end{aligned}$$

$$\{(k, (b_1, c_1))\} \Downarrow \{(k', (b_2, c_2))\} = \begin{cases} \{(k, (b_1 \uplus b_2, c_1 \uplus c_2))\} & \text{if } k = k' \\ \{(k, (b_1, c_1)), (k', (b_2, c_2))\} & \text{otherwise} \end{cases}$$

H_{f_1, f_2}^\uparrow is a binary homomorphism (homomorphism from the product monoid) defined in the following way:

$$\begin{aligned}
H_{f_1, f_2}^\uparrow(X_1 \uplus X_2, Y_1 \uplus Y_2) &= H_{f_1, f_2}^\uparrow(X_1, Y_1) \Downarrow H_{f_1, f_2}^\uparrow(X_2, Y_2) \\
H_{f_1, f_2}^\uparrow(\{x\}, \{y\}) &= f_1(x) \Downarrow f_2(y)
\end{aligned}$$

Fig. 4. Characterisation of operators as homomorphisms of the form H_f^\otimes

Flights.

$$\begin{aligned}
\mu(R, \lambda (X \rightarrow \\
\text{fimap}(\lambda ((\text{corr}, (\text{Flight}(\text{dtime1}, \text{atime1}, \text{dep1}, \text{dest1}, \text{dur1}), \text{Flight}(\text{dtime2}, \text{atime2}, \text{dep2}, \text{dest2}, \text{dur2}))) \rightarrow \\
\text{if } \text{atime1} < \text{dtime2} \text{ then } \{\text{Flight}(\text{dtime1}, \text{atime2}, \text{dep1}, \text{dest2}, \text{dur1} + \text{dur2})\} \text{ else } \{\}), \\
\text{join}(\text{fimap}(\lambda (\text{Flight}(\text{dtime}, \text{atime}, \text{dep}, \text{corr}, \text{dur}) \rightarrow (\text{dest}, \text{Flight}(\text{dtime}, \text{atime}, \text{dep}, \text{corr}, \text{dur}))), X)), \\
\text{fimap}(\lambda (\text{Flight}(\text{dtime}, \text{atime}, \text{corr}, \text{dest}, \text{dur}) \rightarrow (\text{dep}, \text{Flight}(\text{dtime}, \text{atime}, \text{corr}, \text{dest}, \text{dur}))), R))))))
\end{aligned}$$

where R is a dataset of direct flights. $\text{Flight}(\text{dtime}, \text{atime}, \text{dep}, \text{dest}, \text{dur})$ is a flight object with a departure time dtime , arrival time atime , departure location dep , destination dest and duration dur . At each iteration, the fixpoint expression computes new flights by joining the flights obtained at the

previous iteration with the flights dataset, in such a way that two flights produce a new flight if the first flight arrives before the second flight departs, and the first flight destination airport is the second's flight departure airport. The computation stops when no more new non-direct flights can be deduced.

Path planning.

$$\text{fmap}(\lambda ((s, d), l) \rightarrow \text{if } s = \text{"Paris"} \text{ and } d = \text{"Geneva"} \text{ then } \{\text{Path}(s, d, l)\} \text{ else } \{\}), \\ \text{reduceByKey}(\text{bestRated}, \text{fmap}(\lambda ((s, d, l) \rightarrow ((s, d), l))), F))$$

$$F = \mu(R, \lambda (X \rightarrow \text{fmap}(\lambda ((k, ((s, l_1), (d, l_2))) \rightarrow \{(s, d, l_1++l_2)\}), \text{join}(\\ \text{fmap}(\lambda ((s, k, l) \rightarrow \{(k, (s, l))\}), X), \\ \text{fmap}(\lambda ((\text{City}(k, l_1), \text{City}(d, l_2)) \rightarrow (k, (d, l_2))), R))))))$$

where R is a set of routes between two cities. Each city $\text{City}(n, l)$ has a name n and a set of landmarks l and each landmark $\text{Landmark}(n, r)$ has a rating r . $\text{bestRated}(l_1, l_2)$ is a function that returns the best set of landmarks based on its ratings.

The fixpoint F computes the set of landmarks that can be visited for each possible path between each two cities. The final term then computes the best path between Paris and Geneva.

Movie Recommendations.

$$\mu(S, \lambda (X \rightarrow \text{fmap}(\lambda (x \rightarrow \text{fmap}(\lambda (\text{User}(u, bm) \rightarrow \text{if } x \in bm \text{ then } bm \text{ else } \{\}), U)), X)))$$

where U is a set of users, each user $\text{User}(u, bm)$ has a set of best movies bm .

The query computes a set of recommended movies by starting from a set of movies S and by adding the best movies of a user if one of his best movies is in the set of recommended movies until no new movie is added.

2.5 Evaluation of expressions

2.5.1 Local execution.

Pattern matching and function application. The result of matching a value against a pattern is either a set of pattern variable assignments or \perp . It is defined as follows:

$$\begin{aligned} m(v, a) &= \{a \mapsto v\} \\ m(C(v_1, \dots, v_n), C(\pi_1, \dots, \pi_n)) &= m(v_1, \pi_1) \cup \dots \cup m(v_n, \pi_n) \\ m(C(\dots), C'(\dots)) &= \perp \text{ if } C \neq C' \end{aligned}$$

where we extend \cup so that $\perp \cup S = \perp$.

A lambda expression $f = \lambda (\pi_1 \rightarrow e_1 \mid \dots \mid \pi_n \rightarrow e_n)$ contains a number of patterns together with return expressions. When this lambda expression is applied on an argument v ($f v$), the argument is matched against the patterns in order, until the result of the match is not \perp . Let i be the smallest index such that $m(v, \pi_i) = S \neq \perp$, the result of the application is obtained by substituting the free pattern variables in e_i according to the assignments in S .

Monoid homomorphisms. The definition of algebraic operations as monoid homomorphism suggests that they can be evaluated in the following way: $H_f^\otimes(\{v_1\} \uplus \{v_2\} \uplus \dots \uplus \{v_n\}) \rightsquigarrow f(v_1) \otimes f(v_2) \otimes \dots \otimes f(v_n)$. As monoid operators are associative, parts of an expression in the form $e_1 \otimes e_2 \otimes \dots \otimes e_n$ can be evaluated in any order and in parallel.

Fixpoint operator. The fixpoint operator can be evaluated as a loop. To evaluate $\mu(R, \varphi)$, first $\varphi(R)$ is computed. If $\varphi(R)$ is included in R , in the sense of set inclusion, then the computation terminates and the result is the distinct values of R (returning distinct values of R is especially needed if the computation terminates at the first iteration). If not, then the values from $\varphi(R)$ are added to R (using \cup) and we loop by evaluating $\mu(R \cup \varphi(R), \varphi)$. This is summarised by the following reduction rules:

$$\mathcal{R}_{stop} \frac{\varphi(R) \subset R}{\mu(R, \varphi) \rightsquigarrow \text{distinct}(R)} \qquad \mathcal{R}_{loop} \frac{\varphi(R) \not\subset R}{\mu(R, \varphi) \rightsquigarrow \mu(R \cup \varphi(R), \varphi)}$$

Note: The fixpoint operation can be computed in a more efficient way by iteratively applying φ to only the new values generated in the previous iteration. This terminates when no new values are added. Applying φ to only the new values and not to the whole set is correct thanks to the homomorphism property of φ (we have $\varphi(X \cup \varphi(X)) = \varphi(X \cup (\varphi(X) \setminus X)) = \varphi(X) \cup \varphi((\varphi(X) \setminus X))$). The algorithm is as follows:

```

1 res = R
2 new = R
3 while new ≠ ∅:
4     new = φ(new) \ res
5     res = res ∪ new
6 return res

```

2.5.2 Distributed execution. We consider in a distributed setting that distributed bags are partitioned. Distributed data is noted in the following way: $R = R_1|R_2|\dots|R_p$, meaning that R is split into p partitions stored on p machines. We can write a new slightly different version of the rule described above for evaluating partitioned data:

- $H_f^\cup(R_1|R_2|\dots|R_p) \rightsquigarrow H_f^\cup(R_1)|H_f^\cup(R_2)|\dots|H_f^\cup(R_p)$ (partitioning does not have to change)
- $H_f^\otimes(R_1|R_2|\dots|R_p) \rightsquigarrow H_f^\otimes(R_1) \otimes_{nl} H_f^\otimes(R_2) \otimes_{nl} \dots \otimes_{nl} H_f^\otimes(R_p)$, where \otimes_{nl} is the non-local version of \otimes . Applying this non-local operation means that data transfers are required.

This means that in our algebra, all operators apart from flatmap (which is a homomorphism H_f^\cup) need to send data across the network (for executing the non-local version of their monoid operator). The execution of these non-local operators depends on the distributed platform. Spark for example performs *shuffling* to redistribute the data across partitions for the computation of certain its operations like cogroup and groupByKey.

3 OPTIMIZATIONS

In this section, we propose new optimization rules for terms with fixpoints, and describe when and how they apply. The purposes of the rules are (i) to identify which basic operations within an algebraic term can be rearranged and under which conditions, and (ii) to describe how new terms are produced or evaluated after transformation.

We first give the intuition behind each optimization rule before zooming on each of them to formally describe when they apply. The four new optimization rules are:

- PF is a rewrite rule of the form:

$$F(\mu(R, \varphi)) \longrightarrow \mu(F(R), \varphi)$$

it aims at pushing a filter F inside a fixpoint, whenever this is possible. A filter is a function which keeps only some elements of a dataset based on their values; we define it formally in Sec. 3.1.1.

- PJ is a rewrite rule of the form:

$$\text{join}(A, \mu(R, \varphi)) \longrightarrow \text{join}(A, \mu(F_A(R), \varphi))$$

it aims at inserting a filter F_A inside a fixpoint before a join is performed. It is inspired by the semi-join found in relational databases, and tailored for μ -monoids.

- PA is a rewrite rule of the form:

$$f(\mu(R, \varphi)) \longrightarrow f(\mu(R, f \circ \varphi))$$

it aims at pushing a function f (such as reduce) inside a fixpoint. It is inspired from the premapability condition in Datatalog [26].

- an optimization rule P_{dist} that determines how a fixpoint term is evaluated in a distributed manner by choosing among two possible execution plans.

3.1 Pushing filter inside a fixpoint (PF)

3.1.1 Filter depending on a single pattern variable.

Definition 3.1 (filter). We call *filter* a function of the form $\lambda (D \rightarrow \text{flmap}(\lambda (\pi \rightarrow \text{if } c(a) \text{ then } \{\pi\} \text{ else } \{\}) , D))$, where π is a pattern containing the variable a and $c(a)$ is a Boolean condition depending on the value of a .

Such a function returns the dataset D filtered by retaining only the elements whose value for a (as determined by pattern-matching that element with π) satisfies $c(a)$. The elements are unmodified, so the result is a subcollection of D .

In the following, we consider a filter F with π and a defined as above, and we denote by π_a the function that matches an element against π and returns the value of a ($\pi_a = \lambda (\pi \rightarrow a)$). For instance, $\pi_a((1, (5, 6))) = 5$ for $\pi = (x, (a, y))$.

Let us consider a term $A = F(D)$. In terms of denotational semantics, with the notations above, we have $A = \{d \in D \mid c(\pi_a(d))\}$.

The PF rule. This rule consists in transforming an expression of the form $F(\mu(R, \varphi))$ to an expression of the form $\mu(F(R), \varphi)$, where F is a filter.

In the second form, the filter is pushed before the fixpoint operation. In other words, the constant part R is filtered first before applying the fixpoint on it. We now present sufficient conditions for the two terms to be equivalent.

PF condition. Let (C) be the following condition:

$$\forall r \in R \quad \forall s \in \varphi(\{r\}) \quad \pi_a(r) = \pi_a(s)$$

We can show (see Appendix B.3) that if (C) is satisfied, then the filter can be pushed.

Intuitively, this condition means that the operation φ does not change the part of its input data that corresponds to a in the pattern π , which is the part used in the filter; so for each record in the fixpoint that does not pass the filter, the record in R that has originated it does not pass the filter and the other way round. That is why we can just filter R in the first place.

Verifying the condition (C) using type inference. We will start by explaining the intuition behind this before going into the details.

For the condition (C) to hold, we need to make sure that the part of the data extracted by π_a is not modified by φ . For this, our solution is based on the following observation: if f is a polymorphic function whose argument contains exactly one value of the undetermined type α and whose result must also contain a value of type α , then the α value in the result is necessarily the one in the argument ($f : \alpha \rightarrow \alpha \Rightarrow \forall x f(x) = x$).

This reasoning can also be used for a more complex input type $C(\alpha)$ that contains a polymorphic type. For instance: $C(\alpha) = A(B(\alpha), D)$ is such a type given that A, B and D are type constructors. So our goal is, given that φ takes as input a bag of elements of type C , to find an appropriate polymorphic type $C(\alpha)$ that will be used for type checking φ . In practice, we translate the φ operation to a Scala function that takes a polymorphic input type and use the Scala type inference system [18] to get the output type⁸. $C(\alpha)$ should be built in such a way that the position of α in $C(\alpha)$ is the same as the position of a in π . Such a type is possible to build because the type C matches the pattern π , otherwise the filtered term would not be type correct (see Appendix A). Finally, if the output type also contains the type α and has the same position as a in π then we can show that the condition (C) holds.

Building $C(\alpha)$. Types are made from type constructors and basic types, and patterns are made from type constructors and pattern variables. So we can represent their structures using trees. In the following we sometimes refer to types by the trees representing them.

Definition 3.2 (path). We define the *path* to the node labelled n in the tree T denoted $\text{path}(n, T)$ by the ordered sequence $\text{Seq}(a_i)$ where a_i is the next child arity of the i th visited node to reach n from the root of the tree. A node in a tree can be identified by its path.

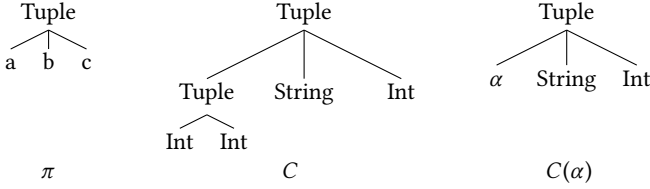
Let us consider the function $\text{replace}_\alpha(p, T)$ that, given a path p and a type T returns a polymorphic type $T(\alpha)$ that is obtained by replacing in T the node at path p and its children by a node labelled α . Let us now consider $C(\alpha) = \text{replace}_\alpha(\text{path}(a, \pi), C)$, where $\text{Bag}[C]$ is the input type of φ . Note that this path makes sense in C because C matches π (see Appendix A).

With $C(\alpha)$ built this way, we have the following:

$$e : C \text{ and } \pi_a(e) : \alpha \Rightarrow e : C(\alpha) \quad (1)$$

$$e : C(\alpha) \Rightarrow \pi_a(e) : \alpha \quad (2)$$

For example:



We show that if $\varphi : \text{Bag}[C(\alpha)] \rightarrow \text{Bag}[C(\alpha)]$ then the condition (C) is verified:

Let $r \in R$ and let us take $\alpha = \{\pi_a(r)\}$ which is the singleton type containing the value $\pi_a(r)$. Since $\pi_a(r) : \alpha$, we have $r : C(\alpha)$ according to (1).

We also have $\varphi(\{r\}) : \text{Bag}[C(\alpha)]$ because $\{r\} : \text{Bag}[C(\alpha)]$ and $\varphi : \text{Bag}[C(\alpha)] \rightarrow \text{Bag}[C(\alpha)]$ which means that $\forall s \in \varphi(\{r\}) \quad s : C(\alpha)$. So $\pi_a(s) : \alpha$ (according to (2)) so $\pi_a(s) = \pi_a(r)$ hence (C).

3.1.2 Filters depending on multiple variables. We showed that (C): $\forall r \in R \quad \forall s \in \varphi(\{r\}) \quad \pi_a(r) = \pi_a(s)$ is sufficient for pushing the filter in a fixpoint when the filter condition depends on a . We can easily show that when the filter depends on a set of pattern variables V , the sufficient condition becomes: $\forall r \in R \quad \forall s \in \varphi(\{r\}) \quad \forall v \in V \quad \pi_v(r) = \pi_v(s)$. So, if one of the variables in V does not satisfy the condition the filter would not be pushed. However, we can do better by trying to split the condition c to two conditions c_1 and c_2 , such that $c = c_1 \wedge c_2$ and c_1 depends only on the subset of variables that satisfies the condition (this splitting technique is used in [10] to push filters in a cogroup or a groupby). If such a split is found, the filter $\text{fmap}(\lambda (\pi \rightarrow \text{if } c \text{ then } \{\pi\} \text{ else } \{\}),$

⁸We consider it a more practical solution than implementing our own type inference system supporting polymorphism.

R) can be rewritten as $\text{fmap}(\lambda (\pi \rightarrow \text{if } c_2 \text{ then } \{\pi\} \text{ else } \{\}), \text{fmap}(\lambda (\pi \rightarrow \text{if } c_1 \text{ then } \{\pi\} \text{ else } \{\}), R)$. The inner filter can then be pushed.

3.2 Filtering inside a fixpoint before a join (PJ)

Let us consider the expression: $\text{join}(A, B)$, where A is a constant and $B = \mu(R, \varphi)$. After the execution of the fixpoint, the result is going to be joined with A , so only elements of this result sharing the same keys with A are going to be kept. So in order to optimize this term, we want to push a filter that keeps only the elements having a key in A . This way, elements not sharing keys with A are going to be removed before applying the fixpoint operation on them.

- (1) we show that $\text{join}(A, B) = \text{join}(A, F_A(B))$, where $F_A(B) = \{(k, v) \mid (k, v) \in B, \exists w (k, w) \in A\}$.
- (2) we show that $F_A(B)$ is a filter on B . This filter can be pushed when the criteria on pushing filters is fulfilled.
- (3) we show as well that

$$\forall R, F_A(R) = \text{fmap}(\lambda ((k, (s_v, s_w)) \rightarrow \text{if } s_w \neq \{\} \text{ then } \text{fmap}(\lambda (v \rightarrow \{(k, v)\}), s_v) \text{ else } \{\}), \text{cogroup}(R, A))$$

Proofs of the above are in Appendix B.5.

3.3 Pushing aggregation into a fixpoint (PA)

The PA rule. consists in rewriting a term of the form $f(\mu(R, \varphi))$ to a term of the form $f(\mu(R, f \circ \varphi))$.

PA conditions. Let us consider the following conditions:

- f is a homomorphism: $\cup \rightarrow \oplus$
- f is idempotent. Typically, this is the case if f is a `reduce` or a `reduceByKey`.
- $f \circ \varphi \circ f = f \circ \varphi$.

When these conditions are met⁹, we can rewrite $f(\mu(R, \varphi))$ to $f(\mu(R, f \circ \varphi))$, aggregating progressively at each iteration instead of once after the whole computation (see proof in Appendix B.6).

Applying this optimization on the expression of the SP example (Sec.2.4) means that only the shortest paths are kept at each iteration of the fixpoint so we avoid computing all possible paths before keeping only the shortest ones at the end.

3.4 Distribution of the fixpoint operations (\mathbf{P}_{dist})

As explained in 2.5.1, the fixpoint operation is computed locally using a loop (defined by $\mathcal{R}_{\text{loop}}$ and $\mathcal{R}_{\text{stop}}$). To evaluate the fixpoint in a distributed setting, we could simply write a loop that distributes the computation of the operation that is performed at each iteration ($\lambda (X \rightarrow X \cup \varphi(X))$) among the workers. We call this execution plan \mathcal{P}_1 and we define it by the reduction rules $\mathcal{R}_{\text{stop}}$ and the following distributed version of $\mathcal{R}_{\text{loop}}$:

$$\frac{\varphi(R_1|R_2|\dots) \not\subseteq R_1|R_2|\dots}{\mu(R_1|R_2|\dots, \varphi) \rightsquigarrow \mu((R_1 \cup \varphi(R_1)) \cup_{nl} (R_2 \cup \varphi(R_2)) \cup_{nl} \dots, \varphi)}^{10}$$

We recall that $R_1|R_2|\dots$ denotes a distributed bag split across different partitions R_i . \cup_{nl} denotes non local union without duplicates.

⁹As we will explain in 3.5.2, we require in practice an annotation from the programmer to know that these conditions hold.
¹⁰We have $R \cup \varphi(R) = (R_1 \cup \varphi(R_1)) \cup (R_2 \cup \varphi(R_2)) \cup \dots$ (where $R = R_1 \cup R_2 \cup \dots$) because φ is a monoid homomorphism from \cup to \cup

\mathcal{P}_1 performs \cup_{nl} at each iteration to remove duplicates from the iteration results across all partitions. In the TC example (Sec. 2.4), this plan amounts to appending, at each iteration, all currently found paths from all partitions with the graph edges R .

Alternatively, if we use the fact that $\mu(R, \varphi)$ is a monoid homomorphism, then we can apply the following reduction rule to evaluate it:

$$\mu(R_1|R_2|\dots, \varphi) \rightsquigarrow \mu(R_1 \cup \varphi(R_1), \varphi) \cup_{nl} \mu(R_2 \cup \varphi(R_2), \varphi) \cup_{nl} \dots$$

Then each $\mu(R_i, \varphi)$ is going to be evaluated by \mathcal{R}_{loop} and \mathcal{R}_{stop} as they are fixpoints on local bags. This execution plan, that we name \mathcal{P}_2 , will avoid doing non local set unions between all partitions at each iteration of the fixpoint. Instead, the fixpoint is executed locally on each partition on a part of the input, after which set union \cup_{nl} is computed once to gather results. In our example, this amounts to computing, on each partition i , all paths in the graph starting from nodes in R_i , the result is then the union of all obtained paths.

This reduction in data transfers can lead to a significant improvement of performance, since the size of data transfers over the network is a determining factor of the performance of distributed applications.

The optimization rule \mathcal{P}_{dist} uses the plan \mathcal{P}_2 instead of \mathcal{P}_1 for evaluating fixpoints.

Avoiding \cup_{nl} in \mathcal{P}_2 . \mathcal{P}_2 can be optimized further by repartitioning the data in the cluster in such a way that every result of the fixpoint appears in one partition only. When that is the case, it is sufficient to perform a bag union rather than a set union that removes duplicates from across the cluster. If we know that there is a part in the input that does not get modified by φ , we can repartition the data on this part of the input (no two different partitions have the same value for this part), so the result of the fixpoint is also going to be repartitioned in the same way. We formalize this optimization in the following way:

Let π a pattern that matches the input of φ and a a pattern variable in π . We consider the following propositions:

$$(C_a) : \forall r \in R \quad \forall s \in \varphi(\{r\}) \quad \pi_a(r) = \pi_a(s)$$

$$(P_a) : \forall i \neq j \quad \forall x \in R_i \quad \forall y \in R_j \quad \pi_a(x) \neq \pi_a(y)$$

We show (Appendix B.4) that if there exists a pattern variable a that verifies (C_a) , then

$$P_a \Rightarrow \forall i \neq j \quad \mu(R_i, \varphi) \cap \mu(R_j, \varphi) = \emptyset$$

Which means that $\mu(R_1 \cup \varphi(R_1), \varphi) \cup_{nl} \mu(R_2 \cup \varphi(R_2), \varphi) \cup_{nl} \dots = \mu(R_1 \cup \varphi(R_1), \varphi) \mid \mu(R_2 \cup \varphi(R_2), \varphi) \mid \dots$

The pattern variable a that verifies (C_a) can be found by using the technique explained in Section 3.1.1. We explore every node n in C (the input type of φ) starting from the root of C and we build $C(\alpha) = \text{replace}_\alpha(\text{path}(n, C), C)$ until we find a node that verifies $\varphi : C(\alpha) \rightarrow C(\alpha)$.

If such a is found, we repartition the data according to (P_a) by using the API provided by the big data platform on which the code is executed, given that a can be extracted from the input data using pattern matching.

3.5 Rule application criteria

Rule PF is a logical optimization rule in the sense that the term it produces is always more efficient than the initial term. Indeed, a filter reduces the size of intermediate data. The application of PF thus reduces data transfers. Operators are also executed faster on smaller data. The application of PF can thus only improve performance.

The rest of the rules however require specific criteria to ensure that their application actually enhances performance.

3.5.1 Criteria for PJ. The rule PJ introduces an additional cogroup to compute the filter being pushed in the fixpoint (as detailed in Sec. 3.2). To estimate the cost of evaluating a term, two important aspects are considered: the size of non-local data transfers it generates, and the local complexity of the term (i.e. the time needed for executing its local operations). PJ can improve local complexity. The reason is that the additional cogroup is evaluated only once, whereas the pushed filter makes R (the first argument of the fixpoint $\mu(R, \varphi)$) smaller. Therefore, in general, each iteration of the fixpoint is executed faster as it deals with increasingly less data (each value removed from the initial bag would have generated more additional values with each iteration). The final join with the result of the fixpoint also executes faster because its size is reduced prior to the join. We can then consider that, in general, the additional cogroup cost is compensated by the speedup of each iteration in the fixpoint as well as the final join. To analyse the impact of the rule on non-local data transfers, we estimate and compare the size of transfers incurred by the terms: $join(A, \mu(R, \varphi))$ and $join(A, \mu(F_A(R), \varphi))$ (obtained after applying the rule). As mentioned in Section 2.5.2, all our algebraic operators apart from `flatMap` trigger non-local transfers. We then consider the following, where $size_t(e)$ is the size of transfers incurred by the term e :

- We assume that `groupby(A)` incurs a transfer size that is linear to the size of A (all A needs to be sent to be seen by other partitions). So, $size_t(groupby(A)) \approx o(size(A))$
- Similarly, `cogroup(A, B)` and `join(A, B)` transfer A and B (the cogroup and join operators are made between all elements of A and B):
 $size_t(cogroup(A, B)) \approx o(size(A) + size(B))$
 $size_t(join(A, B)) \approx o(size(A) + size(B))$
- $\mu(R, \varphi)$ would have to send all its result in order to compute the set union operation. So we just refer to $size(\mu(R, \varphi))$ to indicate the size of the fixpoint result.

Let $S_1 = size_t(join(A, \mu(R, \varphi)))$

and $S_2 = size_t(join(A, \mu(F_A(R), \varphi)))$

$S_1 \approx o(size(A) + 2 \times size(\mu(R, \varphi)))$, here the result of the fixpoint is sent twice: the first time to compute the fixpoint and the second time to compute the join between A and the fixpoint result.

$S_2 \approx o(2 \times size(A) + 2 \times size(\mu(F_A(R), \varphi)) + size(R))$, here $F_A(R)$ requires making a cogroup between A and R which incurs an additional transfer of their sizes. On the other hand, only a filtered fixpoint result is sent.

In order to determine if PJ improves data transfers we compare S_1 and S_2 , which amounts to comparing the following quantities: $2 \times size(\mu(R, \varphi))$ and $2 \times size(\mu(F_A(R), \varphi)) + size(A) + size(R)$. In other words, this estimates whether the data removed from the fixpoint result (by pushing the filter into it) makes up for the sizes of A and R that are transferred to compute the additional cogroup.

3.5.2 Criteria for PA. The PA rule applies the function f on the fixpoint intermediate results. When f performs an aggregation (such as `reduce` or `reduceByKey`), the size of these results is reduced. This means that the fixpoint operation deals with less data at each iteration (which also generally reduces the number of iterations). For example, if we are computing the shortest paths, applying the rule would mean that we are only going to deal with the shortest paths at each step instead of the entirety of possible paths. This can also lead to the termination of the program in case the graph has cycles (note that the programs are semantically equivalent but the evaluation of the first does not terminate). Additionally, when P_{dist} is applied, PA can only reduce the size of the data transferred across the network because f is executed locally and reduces the sizes of the local fixpoints.

To be applicable, this optimization requires the idempotence of f and the constraint $f \circ \varphi \circ f = f \circ \varphi$ to be verified. The latter constraint means that the application of f first before the φ operation does not impact the result compared to when it is applied once at the end. For instance, if we are

computing the shortest paths between a and b , we look for all paths between a and c , append them to paths from c to b , then keep the shortest ones. Alternatively, we could start by keeping only the shortest paths between a and c then append them to paths between c and b without altering results. At present, we do not have a method for statically checking this constraint. So, in practice, we require an annotation from the programmer on the aggregation operations that verify the necessary constraints.

This rule is then applied whenever the constraints are verified.

3.5.3 Criteria for P_{dist} in the Spark setting. The application of the rule P_{dist} can exploit platform-specific criteria. For instance, for Spark [25], the choice between plans \mathcal{P}_1 and \mathcal{P}_2 is parameterized based on two key aspects. First, for a term $\mu(R, \varphi)$, the collections referenced in φ have to be available locally in each worker so that it can compute the fixpoint locally. For instance, if $\varphi = \text{join}(X, S)$ then S and X (at each iteration) are both referenced by φ . This is a limitation of plan \mathcal{P}_2 : when those datasets become too large to be handled by one worker, \mathcal{P}_1 is favored. Second, in Spark, a factor that determines the efficiency of \mathcal{P}_2 is the number of partitions used by the program. Increasing the number of partitions increases the parallelization and reduces the load on each worker because the local fixpoints start from smaller constant parts. For a term $\mu(R, \varphi)$, it is thus possible to regulate the load on the workers by splitting R into smaller R_i , resulting in smaller tasks on more partitions. The ideal number of partitions is the smallest one that makes all workers busy for the same time period, and for which the size of the task remains suitable for the capacity of each worker. Increasing the number of partitions further would only increase the overhead of scheduling. Thus, before choosing plan \mathcal{P}_2 , the rule P_{dist} estimates an appropriate number of partitions, based on an estimated size of the constant part, the size of intermediate data produced by the fixpoint and the workers memory capacity.

4 EXPERIMENTAL RESULTS

Methodology. We experiment the μ -monoids approach in the context of the Spark platform [25].

We evaluate Spark programs generated from optimized μ -monoids expressions, and compare their performance with the state-of-the-art implementations Emma[1] and DIQL [10], which are Domain Specific Languages (more detail about them in Section 5). The authors of Emma showed that their approach outperforms earlier works in [1]. DIQL is a DSL built on monoid algebra (of which the mu-monoids algebra is an extension). Comparing against DIQL shows the interest of having a first-class fixpoint operator in the monoid algebra.

The expressions considered in these experiments are the ones presented in the examples (Section 2.4). The programs generated by μ -monoids from these expressions were obtained by systematically applying the rules PF, PJ, PA, P_{dist} (of Section 3). We evaluate these programs by comparing their execution times against the following programs:

- **DIQL:** The examples have been expressed using DIQL [10] queries. In particular, the fixpoint operation is expressed in terms of the more generic `repeat` operator of the DIQL language. We have written the queries in such a way that they compute the fixpoint more efficiently using the algorithm mentioned in 2.5.1. All DIQL queries used are given in Appendix ??.
- **Emma:** We used the example provided by Emma authors [16] to compute the TC queries, and we wrote modified versions to compute the SP and the path planning examples.
- **mu-monoid-no-PA** mu-monoid without the application of PA to assess the impact of the PA rule on the SP and the path planning examples.
- **manual-spark:** Hand written Spark program. It uses a loop in the driver to compute the fixpoint. So it is equivalent to the mu-monoid program without the P_{dist} optimization. We use it to assess the impact of this optimization.

| Dataset | Edges | Nodes | TC size |
|---------------|-----------|--------|-------------|
| rnd_0.001_10k | 50,119 | 10,000 | 5,718,306 |
| rnd_0.001_20k | 199,871 | 20,000 | 81,732,096 |
| rnd_0.001_30k | 450,904 | 30,000 | 255,097,974 |
| rnd_0.005_10k | 249,791 | 10,000 | 39,113,982 |
| rnd_0.001_40k | 799,961 | 40,000 | 531,677,274 |
| rnd_0.001_50k | 1,250,922 | 50,000 | 906,630,823 |

| Dataset | Edges | Nodes |
|---------------|------------|------------|
| Yago [11] | 62,643,951 | 42,832,856 |
| Facebook [15] | 88,234 | 4,039 |
| DBLP [15] | 1,049,866 | 317,080 |

Table 1. Synthetic and real graphs used in experiments.

In addition to the examples of section 2.4, we evaluate two variants of TC and SP: TC filter and SP filter, where we compute the paths starting from a subset of 2000 nodes randomly chosen in the graph.

Datasets. We use two kinds of datasets:

- Real world graphs of different sizes, presented in Table 1, including a knowledge graph (the Yago [11] dataset¹¹), a social network graph (Facebook), and a scientific collaborations network (DBLP) taken from [15].
- Synthetic graphs shown in Table 1, generated using the Erdos Renyi algorithm that, given an integer n and a probability p , generates a graph of n vertices in which two vertices are connected by an edge with a probability p . rnd_p_n denotes such a synthetic graph, whereas rnd_p_n_W denotes a rnd_p_n graph with edges weighted randomly (between 0 and 5).

Other synthetic graphs are:

- flight_p_n : where edges are taken from rnd_p_n with random depart and arrival times and duration assigned to them.
- c_p_n : serialized object RDD files representing paths between cities. It is also generated from rnd_p_n , each city has been assigned up to 10 random landmarks.
- u_n : serialized object RDD files of n users, each assigned up to 15 random movies.

Experimental setup. Experiments have been conducted on a Spark cluster composed of 5 machines (hence using 5 workers, one on each machine, and the driver on one of them)¹².

For the Yago dataset, transitive closures are computed for the `isLocatedIn` edge label. The hand written spark program (manual-spark) has the optimizations PF and PA whenever possible, P_{dist} is the only rule it does not have. We have also written the DIQL queries in such a way they apply PF. Such a pre-filtering was not possible for Emma because the programs perform a non linear fixpoint. Trying to write a linear version leads to an exception in the execution. We were not able to write an Emma program that computes movie recommendations. Iterating over a users own movies leads to an exception.

Results summary. Figure 5 presents the obtained results. We observe that the programs generated by mu-monoid systematically outperform the other program versions. The speedup is even more important for programs where PA is applied (SP, SP filter and path planning), especially when combined with P_{dist} .

¹¹We use a cleaned version of the real world dataset Yago 2s [11], that we have preprocessed in order to remove duplicate RDF [6] triples (of the form `<source, label, target>`) and keep only triples with existing and valid identifiers. After preprocessing, we obtain a table of Yago facts with 83 predicates and 62,643,951 rows (graph edges).

¹²Each machine has 40 GB of RAM, 2 Intel Xeon E5-2630 v4 CPUs (2.20 GHz, 20 cores each) and 66 TB of 7200 RPM hard disk drives, running Spark 2.2.3 and Hadoop 2.8.4 inside Debian-based Docker containers.

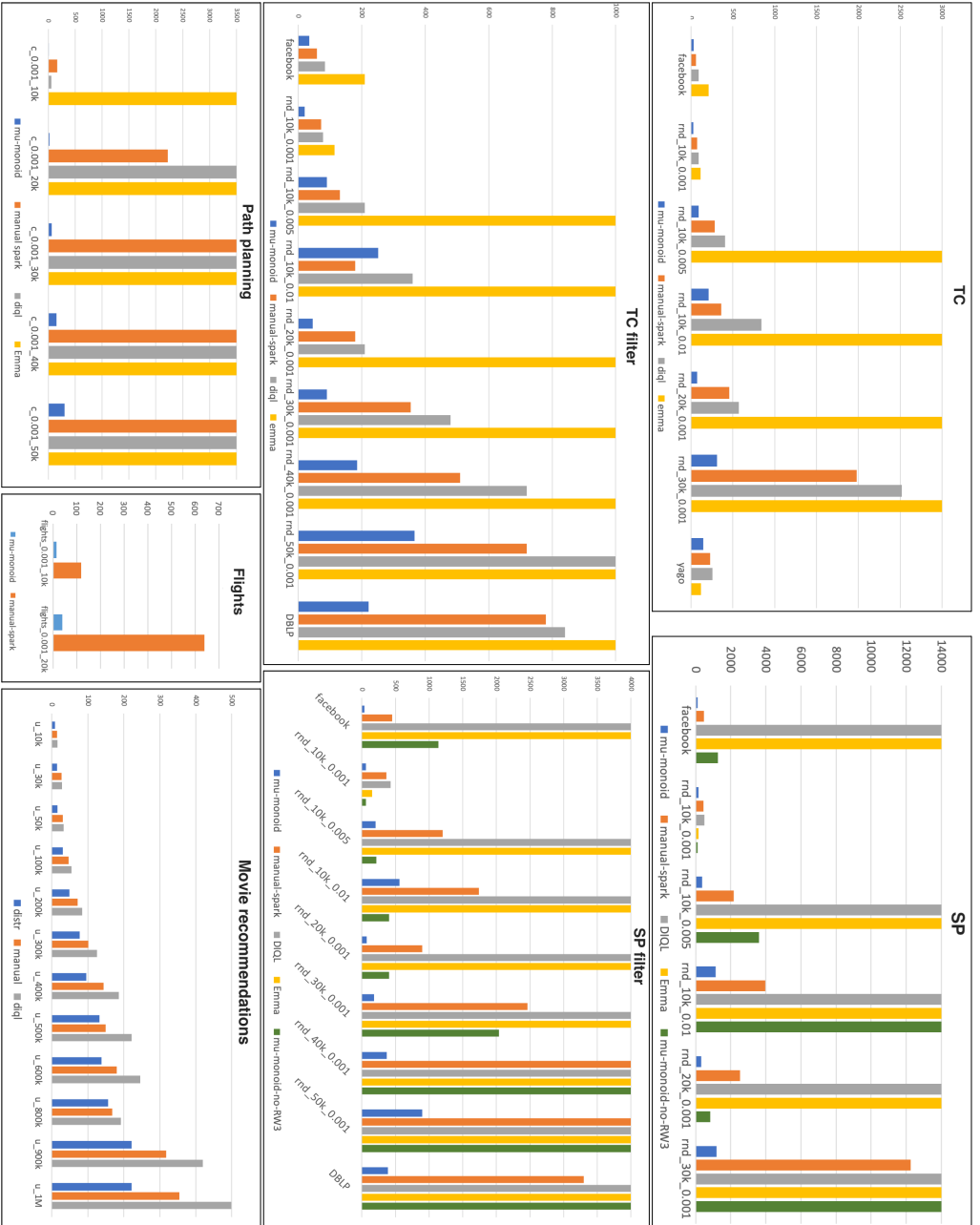


Fig. 5. Comparison of program running times (reported in seconds on the y-axes). A bar reaching the maximal value on the y-axis indicates a timeout.

This experimental comparison shows the benefit of the plan that distributes the fixpoint. It also highlights the benefits of the approach that synthesises code: generating programs that are not natural for a programmer to write, like the distributed loop to compute the fixpoint.

5 RELATED WORKS

Bigdata frameworks such as Spark and Flink offer an API with operations (such as map and reduce) that can be seen as a highly embedded domain specific language (EDSL). As explained in [3], this API approach (as well as that of other EDSLs) offers a big advantage over approaches like relational query languages and Datalog as they allow to express (1) more general purpose computations on (2) more complex data in their native format (which can be arbitrarily nested). Also, functions such as map and reduce can have as argument any function f of the host language¹³ (called second order functions map f and reduce f) thus exposing parallelism while allowing a seamless integration with the host language. However, as pointed out by [3], this approach suffers from the difficulty of automatically optimizing programs. To enable automatic optimizations, they propose an algebra based on monads and monad comprehensions and propose an EDSL called Emma. Emma targets JVM-based parallel dataflow engines (such as Spark and Flink). However, in order to support recursive programs (a large class of programs), one needs to use loops to mimick fixpoints but optimizations are not available for such constructs.

LINQ [17] and Ferry [13] are other comprehension based languages. However, unlike Emma, they do not analyse comprehensions to make optimizations. Moreover, as they target relational database management systems (RDBMS), the set of host language expressions that can be used in query clauses like selection and projection is restricted. A complete survey of those works and of other EDSLs can be found in [3]. The support of recursive query optimization in RDBMS has been recently substantially improved in [14]. However, [14] is restricted to the centralized setting, and to relational algebra. Datalog, another recursive query language has been studied in the distributed setting in [20]. These lines of works focus only on modeling data access (with no or poor support for user-defined functions for instance), not general computations such as in more complete programming languages.

The authors of [1] trace the effort of using monads back to Buneman who showed in [4] that the so-called monads can be used to generalize nested relational algebra to different types of collections and complex objects. The idea of using monoids and monoid homomorphisms for modeling computations with data collections can be even found earlier in the works of [21, 22]. It can also be found in the concepts of on list comprehensions [19, 23], monad comprehensions [24], ringad comprehensions [12].

The work found in [8] is pursuing a similar goal to that of Emma, which is to optimize EDSLs that express distributed computations. He proposed an algebra based on monoid homomorphisms therefore with parallelism at its core: an homomorphic operation H on a collection is defined as the application of H on each subpart of the collection, results are then gathered using an associative operator. Distributed collections are modelled using the union representation of bags, and collection elements can be of any type defined in the host language. The authors designed DIQL [10] (a DSL that translates to the monoid algebra). Using reflection of the host language (Scala in this work) and quotations, queries of this DSL can be compiled and type checked seamlessly with the rest of the host language code. In fact, Emma uses the same approach as well.

Fegaras proposed a monoid comprehension calculus first [9] which later evolved in the monoid algebra presented in [8]. The algebra of [8] has a repeat operator which suffers from the same limitations as Emma [3] : no optimization technique is provided.

In this work, we show that having a fixpoint as a first class operator in the algebra (which can be seen as a fold operation in the monad formalism) introduce even further optimization opportunities, integrates well with the other operators and offers a considerable gain in performance in practice.

¹³The *host language*, according to Alexandrov, is the language in which the EDSL is embedded.

6 CONCLUSION

We propose to extend the monoid algebra with a fixpoint operator that models recursion. The extended μ -monoids algebra is suitable for modeling recursive computations with distributed data collections such as the ones found in big data frameworks. The major interest of the “ μ ” fixpoint operator is that, under prerequisites that are often met in practice, it can be considered as a monoid homomorphism and thus can be evaluated by parallel loops with one final merge rather than by a global loop requiring network overhead after each iteration.

We also propose rewriting rules for optimizing fixpoint terms: we show when and how filters can be pushed into fixpoints. In particular, we find a sufficient condition on the repeatedly evaluated term (φ) regardless of its shape, and we present a method using polymorphic types and a type system such as Scala’s to check whether this condition holds. We also propose a rule to prefilter a fixpoint before a join. The third rule allows for pushing aggregation functions inside a fixpoint.

Experiments suggest that: (i) Spark programs generated by the systematic application of these optimizations can be radically different from – and less intuitive – than the input ones written by the programmer; (ii) generated programs can be significantly more efficient. This illustrates the interest of developing optimizing compilers for programming with big data frameworks.

A WELL-TYPED TERMS

We define typing rules for algebraic terms, in order to exclude meaningless terms. In these rules, we use *type environments* Γ which bind variables to types. An environment contains at most one binding for a given variable. We combine them in two different ways:

- $\Gamma \cup \Gamma'$ is only defined if Γ and Γ' have no variable in common, and is the union of all bindings in Γ and Γ' ;
- $\Gamma + \Gamma'$ is defined by taking all bindings in Γ' plus all bindings in Γ for variables not appearing in Γ' . In other words, if a variable appears in both, the binding in Γ' overrides the one in Γ .

Definition A.1 (matching). We first define the environment obtained by *matching* a data type to a pattern by the following:

$$\frac{\frac{\text{match}(a, t) \rightarrow a : t}{\forall i \text{ match}(\pi_i, t_i) \rightarrow \Gamma_i}}{\text{match}(C(\pi_1, \dots, \pi_n), C[t_1, \dots, t_n]) \rightarrow \Gamma_1 \cup \dots \cup \Gamma_n}$$

If, according to these rules, there is no Γ such that $\text{match}(\pi, t) \rightarrow \Gamma$ holds, we say that pattern π is *incompatible* with type t . Note that, with our conditions, a pattern containing several occurrences of the same variable is not compatible with any type and hence cannot appear in a well-typed term, as the typing rules will show.

Definition A.2 (operation + and relation <: on sum types). The operation + on sum types is defined recursively as follows. Let t be a sum type and C a constructor not appearing in t , then:

$$\begin{aligned} t + (t'_1 \parallel \dots \parallel t'_m) &= (t + t'_1) + (t'_2 \parallel \dots \parallel t'_m) \\ t + C[t_1, \dots, t_n] &= t \parallel C[t_1, \dots, t_n] \\ (t \parallel C[t_1, \dots, t_n]) + C[t'_1, \dots, t'_n] &= t \parallel C[t_1 + t'_1, \dots, t_n + t'_n] \end{aligned}$$

The type $t + t'$ is not defined if t or t' is not a sum type, or if they have constructors in common with incompatible type parameters, i. e. type parameters which cannot themselves be combined with +.

We write $t <: t'$ if $t + t' = t'$.

$$\begin{array}{c}
\frac{\Gamma \vdash e_1 : t \rightarrow T_1[t'] \quad \Gamma \vdash e_2 : T_2[t] \quad T_1 = T_2 \vee (T_1 = \text{Bag}_l \wedge T_2 = \text{Bag}_d)}{\Gamma \vdash \text{flmap}(e_1, e_2) : T_2[t']} \\
\\
\frac{\Gamma \vdash e_1 : t \rightarrow t \rightarrow t \quad \Gamma \vdash e_2 : T[t]}{\Gamma \vdash \text{reduce}(e_1, e_2) : t} \qquad \frac{\Gamma \vdash e : T[t \times t']}{\Gamma \vdash \text{groupby}(e) : T[t \times \text{Bag}_l[t']]} \\
\\
\frac{\Gamma \vdash e_1 : t' \rightarrow t' \rightarrow t' \quad \Gamma \vdash e_2 : T[t \times t']}{\Gamma \vdash \text{reduceByKey}(e_1, e_2) : T[t \times t']} \\
\\
\frac{\Gamma \vdash e_1 : T_1[t \times t_1] \quad \Gamma \vdash e_2 : T_2[t \times t_2] \quad T_3 = (\text{if } T_1 = T_2 \text{ then } T_1 \text{ else } \text{Bag}_d)}{\Gamma \vdash \text{cogroup}(e_1, e_2) : T_3[t \times (\text{Bag}_l[t_1] \times \text{Bag}_l[t_2])]} \\
\\
\frac{\Gamma \vdash e_1 : T_1[t \times t_1] \quad \Gamma \vdash e_2 : T_2[t \times t_2] \quad T_3 = (\text{if } T_1 = T_2 \text{ then } T_1 \text{ else } \text{Bag}_d)}{\Gamma \vdash \text{join}(e_1, e_2) : T_3[t \times (t_1 \times t_2)]} \\
\\
\frac{\Gamma \vdash e_1 : T[t] \quad \Gamma \vdash e_2 : T[t] \rightarrow T[t]}{\Gamma \vdash \mu(e_1, e_2) : T[t]} \qquad \frac{\forall i \Gamma \vdash e_i : t_i}{\Gamma \vdash C(e_1, e_2, \dots, e_n) : C[t_1, t_2, \dots, t_n]} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \{e\} : \text{Bag}_l[t]} \\
\\
\frac{t'_1 + \dots + t'_n = t' \quad \text{match}(\pi_i, t'_i) \rightarrow \Gamma'_i \quad \Gamma + \Gamma'_i \vdash e_i : t_i \quad t_1 + \dots + t_n = t}{\Gamma \vdash \lambda (\pi_1 \rightarrow e_1 \mid \dots \mid \pi_n \rightarrow e_n) : t' \rightarrow t} \\
\\
\frac{\Gamma \vdash e_1 : t_1 \rightarrow t' \quad \Gamma \vdash e_2 : t_2 \quad t_2 <: t_1}{\Gamma \vdash e_1 e_2 : t'} \qquad \frac{\Gamma(a) = t}{\Gamma \vdash a : t} \qquad \frac{\text{type}(c) = t}{\Gamma \vdash c : t}
\end{array}$$

Fig. 6. Typing judgements.

Definition A.3 (Well-typed terms). A term e is *well-typed* in a given environment Γ iff $\Gamma \vdash e : \tau$ for some type t , as judged by the relation defined in Figure 6. In these rules, T represents one of Bag_l or Bag_d .

Note that these rules do not give a way to infer the parameter type of a λ expression in general; we assume some mechanism for that in the language \mathcal{L} . An interesting particular case, however, is when a λ expression is used directly as the first parameter of a $\text{flmap}(\cdot, \cdot)$ or $\text{reduce}(\cdot, \cdot)$. We can write the following compound deterministic rule in the case of $\text{flmap}(\cdot, \cdot)$, for example:

$$\frac{\Gamma \vdash e_2 : T_2[t] \quad \text{match}(\pi, t) \rightarrow \Gamma' \quad \Gamma + \Gamma' \vdash e_1 : T_1[t'] \quad T_1 = T_2 \vee (T_1 = \text{Bag}_l \wedge T_2 = \text{Bag}_d)}{\Gamma \vdash \text{flmap}(\lambda (\pi \rightarrow e_1), e_2) : T_2[t']}$$

B ADDITIONAL PROOFS

In the following, we suppose (fc): $\forall A, B \quad \varphi(A \uplus B) = \varphi(A) \uplus \varphi(B)$

B.1 Proving the existence of the fixpoint

We have $\psi = R \uplus \varphi$

We prove that the fixpoint of $f : X \rightarrow X \cup \psi(X)$ exists:

We consider the following bag $F = \uplus_{n \in \mathbb{N}} \varphi^{(n)}(R)$

We have $\varphi(F) = \varphi(\uplus_{n \in \mathbb{N}} \varphi^{(n)}(R)) = \uplus_{n \in \mathbb{N}} \varphi^{(n+1)}(R)$ (fc)

So $\varphi(F) = \uplus_{n \in \mathbb{N}^*} \varphi^{(n)}(R)$

So $\psi(F) = R \uplus \varphi(F) = R \uplus \biguplus_{n \in \mathbb{N}^*} \varphi^{(n)}(R) = F$ (because $\varphi^{(0)}(R) = R$)

So $f(F) = F \cup \psi(F) = F \cup F = D_F$, where $D_F = \text{distinct}(F)$

We can find bags F_1, \dots, F_n , all without duplicates such that $F = D_F \uplus F_1 \uplus \dots \uplus F_n$ (they can be the singletons remaining after removing D_F from F)

We have $f(F) = F \cup \psi(F) = F \cup (R \uplus \varphi(F)) = F \cup R \cup \varphi(D_F \uplus F_1 \uplus \dots \uplus F_n) = F \cup R \cup (\varphi(D_F) \uplus \varphi(F_1) \uplus \dots \uplus \varphi(F_n))$ (fc)

So $f(F) = F \cup R \cup \varphi(D_F) \cup \varphi(F_1) \cup \dots \cup \varphi(F_n)$

We have $\forall i, F_i \subset D_F$ (because F_i do not contain duplicates and are contained in F), so $\varphi(F_i) \subset \varphi(D_F)$ (fc)

Which means $f(F) = F \cup R \cup \varphi(D_F) = D_F \cup R \cup \varphi(D_F)$ (because \cup removes duplicates)

Since $f(F) = D_F$ (as shown earlier), we have $D_F \cup R \cup \varphi(D_F) = D_F$, which means $f(D_F) = D_F$

Hence f has a fixpoint: $D_F = \text{distinct}(\biguplus_{n \in \mathbb{N}} \varphi^{(n)}(R)) = \bigcup_{n \in \mathbb{N}} \varphi^{(n)}(R)$

B.1.1 D_F is the smallest fixpoint of f . Let F be a fixpoint of f . We have $F = f(F) = R \cup \varphi(R) \cup \varphi(F) \cup \varphi^2(F)$ because (fc). So, we have $\text{distinct}(R) \subset F$. Let $n \in \mathbb{N}$ and let us suppose $\text{distinct}(\varphi^{(n)}(R)) \subset F$. Let $x \in \varphi^{(n+1)}(R)$, we have $\exists y \in \varphi^{(n)}(R) \quad f(y) = x$. Since $y \in F$ and (fc), we have $x \in \varphi(F)$, so $\text{distinct}(\varphi^{(n+1)}(R)) \subset \text{distinct}(\varphi(F))$. We also have $\text{distinct}(\varphi(F)) \subset F$, so $\text{distinct}(\varphi^{(n+1)}(R)) \subset F$. Which means $\forall n \in \mathbb{N} \quad \text{distinct}(\varphi^{(n)}(R)) \subset F$, hence $D_F \subset F$.

B.1.2 More properties of fixpoint expressions. We have:

$$\mu(R, \varphi) = \bigcup_{n \in \mathbb{N}} \varphi^{(n)}(R) = \bigcup_{n \in \mathbb{N}} \varphi(\biguplus_{r \in R} \{r\}) = \bigcup_{n \in \mathbb{N}} (\biguplus_{r \in R} \varphi(\{r\})) = \bigcup_{n \in \mathbb{N}} (\bigcup_{r \in R} \varphi(\{r\})).$$

Which means that $\forall a \in \mu(R, \varphi) \quad a \in \varphi^{(n)}(\{r\})$ for some $r \in R$ and $n \in \mathbb{N}$.

Also, $a \in \text{distinct}(\varphi^{(n)}(\{r\}))$ and $\forall n \in \mathbb{N} \quad \forall r \in R \quad \text{distinct}(\varphi^{(n)}(\{r\})) \subset \mu(R, \varphi)$.

B.2 Proving that the fixpoint operator is a homomorphism from \uplus to \cup

We show that $\forall R_1, R_2 \quad \mu(R_1 \uplus R_2, \varphi) = \mu(R_1, \varphi) \cup \mu(R_2, \varphi)$:

$$\begin{aligned} \mu(R_1 \uplus R_2, \varphi) &= \bigcup_{n \in \mathbb{N}} \varphi^{(n)}(R_1 \uplus R_2) \\ &= \bigcup_{n \in \mathbb{N}} (\varphi^{(n)}(R_1) \uplus \varphi^{(n)}(R_2)) \quad (\text{because } \varphi \text{ satisfies (fc)}) \\ &= (\bigcup_{n \in \mathbb{N}} \varphi^{(n)}(R_1)) \cup (\bigcup_{n \in \mathbb{N}} \varphi^{(n)}(R_2)) = \mu(R_1, \varphi) \cup \mu(R_2, \varphi) \end{aligned}$$

B.3 Condition for pushing a filter

Let $A = \text{flmap}(\lambda (\pi \rightarrow \text{if } c(a) \text{ then } \{\pi\} \text{ else } \{\}) , \mu(R, \varphi))$

And let the condition (C) be: $\forall r \in R \quad \forall s \in \varphi(\{r\}) \quad \pi_a(r) = \pi_a(s)$

We note (*) the following proposition: $\forall s \in \mu(R, \varphi) \quad \exists r \in R \quad \pi_a(r) = \pi_a(s)$

(*) is verified when the condition (C) is fulfilled because

$$\forall s \in \mu(R, \varphi) \quad \exists r \in R \quad \exists n \in \mathbb{N} \quad s \in \varphi^{(n)}(\{r\}) \quad (\text{shown in Appendix B.1.2})$$

. We show that $A = \mu(F(R), \varphi)$ when the condition (C) is fulfilled:

(1) $\mu(F(R), \varphi) \subset A$:

$F(R) \subset R \Rightarrow \mu(F(R), \varphi) \subset \mu(R, \varphi)$ (because $\mu(R, \varphi) = \mu(F(R) \uplus R', \varphi) = \mu(F(R), \varphi) \cup \mu(R', \varphi)$ and $\mu(R, \varphi)$ does not contain duplicates)

Let $s \in \mu(F(R), \varphi)$

$\exists r \in F(R) \quad \pi_a(s) = \pi_a(r)$ (*)

So $c(\pi_a(s)) = c(\pi_a(r)) = \text{true}$ (because $r \in F(R)$)

So $s \in \mu(R, \varphi)$ and $c(\pi_a(s))$ is true, then $s \in A$

(2) $A \subset \mu(F(R), \varphi)$:

Let $s \in A$. We have: $s \in \mu(R, \varphi)$ and $c(\pi_a(s)) = \text{true}$

So $\exists r \in R \quad \exists n \in \mathbb{N} \quad \pi_a(s) = \pi_a(r)$ (because $(*)$ and $s \in \varphi^{(n)}(\{r\})$)

So $c(\pi_a(r)) = c(\pi_a(s)) = \text{true}$

So $r \in F(R)$, which means $\text{distinct}(\varphi^{(n)}(\{r\})) \subset \mu(F(R), \varphi)$ (see fixpoint related proofs in Appendix B.1.2), so $s \in \mu(F(R), \varphi)$.

B.4 P_{dist} proof

Let us suppose there exist a pattern variable a for which (C_a) is verified, and let us suppose (P_a) . Let R_i and R_j partitions of R such that $i \neq j$. We have $\forall s \in \mu(R, \varphi) \quad \exists r \in R \quad \pi_a(r) = \pi_a(s)$ because (C_a) as mentioned in Appendix B.3. Which means that for any $x \in \mu(R_i, \varphi)$ and $y \in \mu(R_j, \varphi)$, $\exists r_i \in R_i \quad \exists r_j \in R_j \quad \pi_a(r_i) = \pi_a(x)$ and $\pi_a(r_j) = \pi_a(y)$. We have $\pi_a(r_i) \neq \pi_a(r_j)$ because (P_a) , so $x \neq y$. Hence $\forall i \neq j \quad \mu(R_i, \varphi) \cap \mu(R_j, \varphi) = \emptyset$.

B.5 PJ proofs

B.5.1 Inserting F_A before the join. We have $\text{join}(A, B) = \{(k, (x, y)) \mid (k, x) \in A, (k, y) \in B\}$.

So $(k, (x, y)) \in \text{join}(A, B) \Leftrightarrow (k, x) \in A \wedge (k, y) \in B \Leftrightarrow (k, x) \in A \wedge ((k, y) \in B \wedge \exists w (k, w) \in A) \Leftrightarrow (k, x) \in A \wedge (k, y) \in F_A(B) \Leftrightarrow (k, (x, y)) \in \text{join}(A, F_A(B))$.

B.5.2 F_A is a filter. We can show that $F_A(B) = \text{flmap}(\lambda ((k, v) \rightarrow \text{if } c(k) \text{ then } \{(k, v)\} \text{ else } \{ \}), B)$ where $c(k)$ is the boolean expression that corresponds to the predicate $\exists w (k, w) \in A$.

This expression can be: $c(k) = \text{reduce}(\vee, \text{flmap}(\lambda ((k', a) \rightarrow k == k'), A))$.

Which means that in case φ fulfills the criteria for pushing filters we will have $F_A(B) = F_A(\mu(R, \varphi)) = \mu(F_A(R), \varphi)$.

B.5.3 Rewriting the filter with a cogroup. We show that $F_A(B) = C$ where:

$$C = \text{flmap}(\lambda ((k, (s_x, s_y)) \rightarrow \text{if } s_y \neq \{ \} \text{ then } \text{flmap}(\lambda (x \rightarrow \{(k, x)\}), s_x) \text{ else } \{ \}), \text{cogroup}(B, A))$$

We have:

$$C = \bigsqcup_{(k, (s_x, s_y)) \in \text{cogroup}(B, A)} \bigsqcup_{x \in s_x} (\text{if } s_y \neq \{ \} \text{ then } \{(k, (x, y))\} \text{ else } \{ \})$$

(1) Let $e \in C$. So $\exists (k, (s_x, s_y)) \in \text{cogroup}(B, A)$ such that $\exists x \in s_x \quad e = (k, x)$ and $s_y \neq \{ \}$.

We have $(k, (s_x, s_y)) \in \text{cogroup}(B, A)$, so $s_x = \{v \mid (k, v) \in B\}$, which means that $(k, x) \in B$ because $x \in s_x$. And $s_y \neq \{ \}$ means that $\exists w (k, w) \in A$, so $e = (k, x) \in F_A(B)$.

(2) Let $(k, x) \in F_A(B)$. We have $(k, x) \in B$ and $\exists w (k, w) \in A$. So $k \in \text{keys}(A) \cup \text{keys}(B)$.

Let $s_x = \{v \mid (k, v) \in B\}$ and $s_y = \{v \mid (k, v) \in A\}$, so $(k, (s_x, s_y)) \in \text{cogroup}(B, A)$.

Since $x \in s_x$ and $s_y \neq \{ \}$ (because $\exists w (k, w) \in A$), then $(k, x) \in C$.

B.6 Proving $f(\mu(R, f \circ \varphi)) = f(\mu(R, \varphi))$

We assume that f is a homomorphism: $\cup \rightarrow \oplus$, that f is idempotent, and that $f \circ \varphi \circ f = f \circ \varphi$.

Let $\Phi: X \mapsto X \cup \varphi(X)$ and $\Psi: X \mapsto X \cup f \circ \varphi(X)$. First, we show by induction that for any $n \in \mathbb{N}$ we have $f(\Phi^n(R)) = f(\Psi^n(R))$: suppose that $f(\Phi^n(R)) = f(\Psi^n(R))$ for a given n , then:

$$\begin{aligned}
 f(\Phi^{n+1}(R)) &= f(\Phi^n(R) \cup \varphi(\Phi^n(R))) \\
 &= f(\Phi^n(R)) \oplus f \circ \varphi(\Phi^n(R)) \text{ (f is a homomorphism } \cup \rightarrow \oplus) \\
 &= f(\Phi^n(R)) \oplus f \circ \varphi \circ f(\Phi^n(R)) \text{ (} f \circ \varphi = f \circ \varphi \circ f) \\
 &= f(\Psi^n(R)) \oplus f \circ \varphi \circ f(\Psi^n(R)) \\
 &= f(\Psi^n(R)) \oplus f \circ f \circ \varphi \text{ (} f \circ \varphi = f \circ \varphi \circ f \text{ and f is idempotent)} \\
 &= f(\Psi^n(R) \cup f \circ \varphi(\Psi^n(R))) \\
 &= f(\Psi^{n+1}(R))
 \end{aligned}$$

Now, if the initial fixpoint term $\mu(R, \varphi)$ denotes a finite set (i. e. if the initial computation terminates), then there exists $N \in \mathbb{N}$ such that $\mu(R, \varphi) = \Phi^N(R)$; but then we have $f(\Psi^{N+1}(R)) = f(\Phi^{N+1}(R)) = f(\Phi^N(R)) = f(\Psi^N(R))$. Then:

$$\begin{aligned}
 \Psi^{N+2}(R) &= \Psi^{N+1}(R) \cup f \circ \varphi(\Psi^{N+1}(R)) = \Psi^{N+1}(R) \cup f \circ \varphi \circ f(\Psi^{N+1}(R)) \text{ (because } f \circ \varphi \circ f = f \circ \varphi) \\
 &= \Psi^{N+1}(R) \cup f \circ \varphi \circ f(\Psi^N(R)) \\
 &= \Psi^N(R) \cup f \circ \varphi(\Psi^N(R)) \cup f \circ \varphi(\Psi^N(R)) = \Psi^{N+1}(R)
 \end{aligned}$$

Hence $\Psi^{N+1}(R)$ is the fixpoint $\mu(R, f \circ \varphi)$, and we have $f(\mu(R, f \circ \varphi)) = f(\mu(R, \varphi))$.

REFERENCES

- [1] Alexander Alexandrov, Asterios Katsifodimos, Georgi Krastev, and Volker Markl. 2016. Implicit Parallelism through Deep Language Embedding. *SIGMOD Record* 45, 1 (2016), 51–58. <https://doi.org/10.1145/2949741.2949754>
- [2] Alexander Alexandrov, Georgi Krastev, and Volker Markl. 2019. Representations and Optimizations for Embedded Parallel Dataflow Languages. *ACM Trans. Database Syst.* 44, 1, Article 4 (Jan. 2019), 44 pages. <https://doi.org/10.1145/3281629>
- [3] Alexander Alexandrov, Georgi Krastev, and Volker Markl. 2019. Representations and Optimizations for Embedded Parallel Dataflow Languages. *ACM Trans. Database Syst.* 44, 1, Article 4 (Jan. 2019), 44 pages. <https://doi.org/10.1145/3281629>
- [4] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. 1994. Comprehension Syntax. *SIGMOD Rec.* 23, 1 (March 1994), 87–96. <https://doi.org/10.1145/181550.181564>
- [5] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. <http://sites.computer.org/debull/A15dec/p28.pdf>
- [6] Richard Cyganiak, David Wood, and Markus Lanthaler. 2014. RDF 1.1 Concepts and Abstract Syntax. <https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225>
- [7] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004)*, San Francisco, California, USA, December 6-8, 2004. 137–150. <http://www.usenix.org/events/osdi04/tech/dean.html>
- [8] Leonidas Fegaras. 2017. An algebra for distributed Big Data analytics. *Journal of Functional Programming* 27 (2017), e27. <https://doi.org/10.1017/S0956796817000193>
- [9] Leonidas Fegaras and David Maier. 2000. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.* 25, 4 (2000), 457–516. <http://portal.acm.org/citation.cfm?id=377674.377676>
- [10] Leonidas Fegaras and Md Hasanuzzaman Noor. 2018. Compile-Time Code Generation for Embedded Data-Intensive Query Languages. In *2018 IEEE International Congress on Big Data, BigData Congress 2018, San Francisco, CA, USA, July 2-7, 2018*. 1–8. <https://doi.org/10.1109/BigDataCongress.2018.00008>
- [11] Max Planck Institute for Informatics and Telecom ParisTech University. 2019. YAGO: A high-quality knowledge base. <https://www.mpi-inf.mpg.de/yago-naga/yago/>
- [12] Jeremy Gibbons. 2016. Comprehending Ringads - For Phil Wadler, on the Occasion of his 60th Birthday. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science)*, Vol. 9600. Springer, 132–151. https://doi.org/10.1007/978-3-319-30936-1_7

- [13] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. 2009. FERRY: Database-Supported Program Execution. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 1063–1066. <https://doi.org/10.1145/1559845.1559982>
- [14] Louis Jachiet, Pierre Genevès, Nils Gesbert, and Nabil Layaïda. 2020 (to appear). On the Optimization of Recursive Relational Queries: Application to Graph Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. <https://hal.inria.fr/hal-01673025v5/document> <https://hal.inria.fr/hal-01673025v5/document>.
- [15] Jure Leskovec. 2019. SNAP: Stanford Large Network Dataset Collection. <https://snap.stanford.edu/data/>
- [16] Volker Markl. 2019. Emma is a quotation-based Scala DSL for scalable data analysis. <https://github.com/emmalanguage>
- [17] Erik Meijer, Brian Beckman, and Gavin M. Bierman. 2006. LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. 706. <https://doi.org/10.1145/1142473.1142552>
- [18] Martin Odersky, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, and et al. 2004. *An overview of the Scala programming language*. Technical Report.
- [19] Simon L. Peyton Jones. 1987. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- [20] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1135–1149. <https://doi.org/10.1145/2882903.2915229>
- [21] Val Tannen, Peter Buneman, and Shamim A. Naqvi. 1991. Structural Recursion as a Query Language. In *Database Programming Languages: Bulk Types and Persistent Data. 3rd International Workshop, August 27-30, 1991, Nafplion, Greece, Proceedings*. 9–19.
- [22] Val Tannen, Peter Buneman, and Atsushi Ohori. 1991. Data Structures and Data Types for Object-Oriented Databases. *IEEE Data Eng. Bull.* 14, 2 (1991), 23–27. <http://sites.computer.org/debull/91JUN-CD.pdf>
- [23] D. A. Turner. 2016. *Recursion Equations as a Programming Language*. Springer International Publishing, Cham, 459–478. https://doi.org/10.1007/978-3-319-30936-1_24
- [24] Philip Wadler. 1992. Comprehending Monads. *Mathematical Structures in Computer Science* 2, 4 (1992), 461–493. <https://doi.org/10.1017/S0960129500001560>
- [25] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65. <https://doi.org/10.1145/2934664>
- [26] Carlo Zaniolo, Mohan Yang, Ariyam Das, Alexander Shkapsky, Tyson Condie, and Matteo Interlandi. 2017. Fixpoint semantics and optimization of recursive Datalog programs with aggregates. *Theory Pract. Log. Program.* 17, 5-6 (2017), 1048–1065. <https://doi.org/10.1017/S1471068417000436>