On the Optimization of Iterative Programming with Distributed Data Collections

Sarah Chlyah, Nils Gesbert, Pierre Genevès, Nabil Layaïda

On the Optimization of Iterative Programming with Distributed Data Collections

Sarah Chlyah, Nils Gesbert, Pierre Genevès, and Nabil Layaïda

Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP*, LIG, 38000 Grenoble, France

Abstract. 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 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 allows 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. Spark performs the join and distinct operations by transferring the datasets (arguments of the operations) accross 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 accross the cluster for distinct. Hence,

* Institute of Engineering Univ. Grenoble Alpes

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.

```
def shortestPaths(R:RDD[(Int,Int,Int)]) = {
  var ret = R
  var X: RDD[(Int, Int, Int)] = R
  var new_cnt = ret.count()
  var cnt = new_cnt
  do {
    cnt = new_cnt
    X = X.map({case (x,y,l1) => (y,(x,l1)) })
      .join(R.map({ case (z,t,l2) => (z,(t,l2)) }))
      .map({case (_,((x,l1),(t,l2))) => (x,t,l1+l2) })
      .subtract(ret)
    ret = ret.union(X).distinct()
    new_cnt = ret.count()
  } while (new_cnt > cnt)
  ret.map({case (x,y,l) => ((x,y),l)}).reduceByKey(min)
}
```

Fig. 1: Shortest paths program.

Another possible optimization is to put the `reduceByKey` operation inside the loop to only keep 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 allow the representation of a program in terms of algebraic operators that can be analysed and transformed so as to produce a program that executes faster. Transformation-

```
def shortestPaths(R:RDD[(Int,Int,Int)]) = {
  val dictR = LocalOps.to_dict(((x:(Int,Int,Int)) => x._1),
    (x:(Int,Int,Int)) => x, sc.broadcast(R.collect()).value)
  var r = R.mapPartitions(part => {
    var ret = part.toList
    var X = ret
    var cnt = ret.size
    var new_cnt = cnt
    do {
      count = new_count
      X = LocalOps.join(LocalOps.to_dict(((x:(Int,Int,Int)) => x._2),
          (x:(Int,Int,Int)) => x, X), dictR)
        .map({case (k, ((x,y,l), (a,b,m))) => (x,b,l+m)}) diff ret
      ret = (ret ++ X).distinct
      new_count = ret.size
    } while (new_cnt > cnt)
    ret.toIterator
  })
  r.distinct().map({case (x,y,l) => ((x,y),l)}).reduceByKey(min)
}
```

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

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 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 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 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 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 driver, 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 allows 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 algebra introduced in [10,8] that we extended with an operator for expressing recursion. This monoid algebra is able to model a subset of a host language (Scala in our case), that expresses computations on distributed platforms (Spark in our case).

Contributions. Our contributions are the following:

1. An extension of the monoid algebra with a fixpoint operator. This allows expressing 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 host language (Scala in our case) subset that is used for computations on a big data framework (through an API provided by the framework). 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 our 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 our analysis allows us to abstract from the host language. This way, constructs of the host language other than those which we model explicitly are represented as *constants* c, as they are going to be left to the host language 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 $t_1 \to t_2$, 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 implement in a distributed context but we can assume the host language provides a `distinct` operation which removes all duplicates from a bag.

In order to allow 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:

$$\begin{array}{lll} v ::= c & & \text{constant} \\ \quad | \;\; C(v_1, v_2, ..., v_n) & & n\text{-ary constructor} \\ \quad | \;\; \{v_1\} \uplus \{v_2\} \uplus ... \uplus \{v_n\} & & \text{bag} \end{array}$$

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 host language value (in particular any function) that is not explicitly defined in our syntax.

We define the following syntax for types:

$$\begin{array}{lll} t_l ::= & & \text{local type} \\ \quad \mathbb{B} & & \text{basic type} \\ \quad | \;\; C_1[t_l, ..., t_l] \;||\; \cdots \;||\; C_n[t_l, ..., t_l] & & \text{sum type} \\ \quad | \;\; \mathsf{Bag}_l[t_l] & & \text{loc. bag type} \end{array} \qquad \begin{array}{lll} t ::= & & \text{type} \\ \quad | \;\; t_l & & \\ \quad | \;\; \mathsf{Bag}_d[t_l] & & \text{dist. bag type} \\ \quad | \;\; t \to t & & \text{func. type} \end{array}$$

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

We also define product types $t_1 \times \cdots \times t_n$ as syntactic sugar for $\mathsf{Tuple}[t_1, ..., t_n]$.

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

For a given type t, we denote by $\mathsf{Bag}_l[t]$ the type of a local bag and by $\mathsf{Bag}_d[t]$ 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 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: $\mathsf{Bag}[t] ::= \mathsf{Bag}_l[t] \mid \mathsf{Bag}_d[t]$. For example, the bag union operator $\uplus : \mathsf{Bag}[t] \times \mathsf{Bag}[t] \to \mathsf{Bag}[t]$ 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), we use a vertical bar | to indicate nonlocal union of local bags into a distributed one.

2.2 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 $\mathsf{flmap}(\lambda\ (a \to \{a+1\}), b)$ has two arguments: a λ-expression $\lambda\ (a \to \{a+1\})$ (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:

$\pi ::= a \mid C(\pi_1, \pi_2, ..., \pi_n)$	pattern: variable, constructor pattern
$e ::= c \mid a \mid \{e\}$	expression: constant, variable, singleton
$\mid\ \lambda\ (\pi_1 \to e_1 \mid \cdots \mid \pi_n \to e_n)$	function with pattern matching
$\mid\ e\ e \mid C(e_1, e_2, ..., e_n)$	application, constructor expression
$\mid\ \mathsf{flmap}(e, e) \mid \mathsf{reduce}(e, e) \mid \mathsf{groupby}(e)$	flatmap, reduce, group by key
$\mid\ \mathsf{reduceByKey}(e, e) \mid \mathsf{cogr}(e, e) \mid \mathsf{join}(e, e)$	reduce by key, cogroup, join by key
$\mid\ \mu\ (e, e)$	fixpoint

To this, we add the following as syntactic sugar:

- $(e_1, ..., e_n)$ with no constructor is an abbreviation for: $\mathsf{Tuple}(e_1, ..., e_n)$
- $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ is an abbreviation for: $\lambda\ (\mathsf{True} \to e_1 \mid \mathsf{False} \to e_2)\ e$
- Constants c can also represent functions (defined in the host language). 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:

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

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 the host language, 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:

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

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 [6].

2.3 μ-monoids denotational semantics

Monoid homomorphisms The semantics of the μ-monoids algebra extends the semantics of the monoid algebra [8]. We first recall basic definitions for 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 $(\mathsf{List}[\alpha], ++, [\,])$ (algebraically called the free monoid on α), whose identity element is the empty list. Let $U_{\mathsf{List}} : \alpha \rightarrow \mathsf{List}[\alpha]$ 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 $H_f^{\otimes} : \mathsf{List}[\alpha] \rightarrow S$ and $H_f^{\otimes} \circ U_{\mathsf{List}} = f$. This homomorphism can be simply defined by:

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

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

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

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 $\mathsf{List}[\alpha]$ by commutativity is isomorphic to $(\mathsf{Bag}[\alpha], \uplus, \{\})$, 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, $(\mathsf{Set}[\alpha], \cup, \emptyset)$, obtained by quotienting with both commutativity and the graphic identity[1].

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 $(\beta, \otimes, e)$ a monoid noted $(\otimes)$, where α and β are arbitrary types. *Suppose $\otimes$ obeys all the algebraic laws of $\oplus$,* then for any function $f : \alpha \to \beta$, there exists a unique homomorphism[2] $H_f^{\otimes} : T[\alpha] \to \beta$ 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 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).

To summarise the earlier definitions in the case of bags, if $(\beta, \otimes, e)$ is a commutative monoid and $f : \alpha \to \beta$ is any function, the monoid homomorphism $H_f^{\otimes}$ from $\uplus$ to $\otimes$ satisfies the following:

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

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

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

[2] It 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.

$$\mathsf{flmap}(f, A) = \biguplus_{a \in A} f(a)$$

$$\mathsf{reduce}(f, A) = r_N, \text{ where } A = \{a_1, a_2, ..., a_N\}, r_n = f(r_{n-1}, a_n), r_1 = a_1$$

$$\mathsf{groupby}(A) = \{(k, \{v \mid (k, v) \in A\}) \mid k \in keys(A)\}$$

$$\mathsf{reduceByKey}(f, A) = \{(k, \mathsf{reduce}(f, \{v \mid (k, v) \in A\})) \mid k \in keys(A)\}$$

$$\mathsf{cogr}(A, B) = \{(k, (\{v \mid (k, v) \in A\}, \{w \mid (k, w) \in B\})) \mid k \in keys(A) \cup keys(B)\}$$

$$\mathsf{join}(A, B) = \{(k, (v, w)) \mid (k, v) \in A \wedge (k, w) \in B\}$$

$$\mu(R, \varphi) = \bigcup_{n \in \mathbb{N}} \Phi^{(n)}(R), \text{ where } \Phi \colon X \mapsto X \cup \varphi(X)$$

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

Fig. 3: Denotational semantics

and groupBy. However we prefer to include them all in the main syntax for clarity.

We explain in appendix B [6] how these operations (except the fixpoint) are monoid homomorphisms which can be defined as $H_f^{\otimes}$ for appropriate f and $\otimes$. As an example, we show it below for the case of groupby, then we give more details about our fixpoint operator $\mu(R, \varphi)$.

groupby operator 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, $\mathsf{groupby}(\{(1,2),(1,4),(2,2),(2,1),(1,3)\}) = \{(1,\{2,4,3\}),(2,\{2,1\})\}$

groupby is a monoid homorphism $H_f^{\uparrow}$ from $(\mathsf{Bag}[\alpha \times \beta], \uplus, \{\})$ to $(\mathsf{Bag}[\alpha \times \mathsf{Bag}[\beta]], \uparrow, \{\})$, where:

$$f \colon \alpha \times \beta \to \mathsf{Bag}[\alpha \times \mathsf{Bag}[\beta]]$$
$$(k, v) \mapsto \{(k, \{v\})\}$$

and the operator $\uparrow$ is defined such that:

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

Note: Because of the way the function f is defined, the groupby operation supports bags of arbitrary types.

Fixpoint operator Let us first comment on the choice of our fixpoint operator. We could define mathematically a more generic operation $\mu(\psi)$ as the smallest fixpoint of the function $\Psi\colon X \to X \cup \psi(X)$. $\mu(\psi)$ then consists on recursively applying Ψ at each iteration on the result of the previous iteration starting from $X = \emptyset$ until the fixpoint is reached and no more new results are added.

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

It can be shown (see Appendix C.1 [6]) that when $\psi = R \uplus \varphi$ for some R, where φ is a monoid homomorphism from $\uplus$ to $\uplus$ $(\uplus \to \uplus)$, the fixpoint exists (Ψ has a fixpoint) and can be reached from the successive application of φ starting from the empty bag $\{\}$. We thus restrict our language to this particular kind of fixpoint, and we use the syntax $\mu(R, \varphi)$ to denote $\mu(R \uplus \varphi)$.

Under this criteria, the μ operator can be seen as a monoid homomorphism $H_f^{\cup}$ from $\uplus$ to $\cup$, where $f(a) = \mu(\{a\}, \varphi)$:

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

Note: The results on fixpoints presented in this paper are valid for any $\mu(R, \varphi)$ where φ is $\uplus \to \uplus$. However, we are currently able to statically verify this criteria for only a subset of such homomorphisms. Specifically, we require φ to be of the form $\lambda\ (X \to T(X))$ where $T(X)$ is defined as follows:

$$\begin{array}{lll} T(X) ::= & & \\ & X & \\ \mid & \mathsf{flmap}(f, T(X)) & X \text{ does not appear in } f \\ \mid & \mathsf{join}(T(X), A) & X \text{ does not appear in } A \\ \mid & \mathsf{join}(A, T(X)) & X \text{ does not appear in } A \end{array}$$

2.4 Examples

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

Transitive closure *(TC)*

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

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 $\mathsf{join}(\mathsf{flmap}(\lambda\ ((a,b) \to \{(b,a)\}), X),\ R)$ 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)

$$\begin{array}{l}\mathsf{reduceByKey}(\min,\\ \quad \mu(R, \lambda\ (X \rightarrow \mathsf{flmap}(\lambda\ ((b, ((a, l_1), (c, l_2))) \rightarrow \{((a, c), l_1 + l_2)\}),\\ \quad\quad \mathsf{join}(\mathsf{flmap}(\lambda\ (((a, b), l_1) \rightarrow \{(b, (a, l_1))\}), X), \mathsf{flmap}(\lambda\ ((b, c, l_2) \rightarrow \{(b, (c, l_2))\}), R)))))))\end{array}$$

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.

Flights

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

where R is a dataset of direct flights. $\mathsf{Flight}(\mathsf{dtime}, \mathsf{atime}, \mathsf{dep}, \mathsf{dest}, \mathsf{dur})$ 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

$$\begin{array}{l}\mathsf{flmap}(\lambda\ (((s, d), l) \rightarrow \mathsf{if}\ s = \text{"Paris"}\ \text{and}\ d = \text{"Geneva"}\ \mathsf{then}\ \{\mathsf{Path}(s, d, l)\}\ \mathsf{else}\ \{\}),\\ \quad\quad \mathsf{reduceByKey}(\mathsf{bestRated}, \mathsf{flmap}(\lambda\ ((s, d, l) \rightarrow ((s, d), l))), F)))\end{array}$$

$$\begin{array}{l}F = \mu(R, \lambda\ (X \rightarrow \mathsf{flmap}(\lambda\ ((k, ((s, l_1), (d, l_2))) \rightarrow \{(s, d, l_1 \mathsf{++} l_2)\}), \mathsf{join}(\\ \quad\quad\quad\quad\quad\quad \mathsf{flmap}(\lambda\ ((s, k, l) \rightarrow \{(k, (s, l))\}), X),\\ \quad\quad\quad\quad\quad\quad \mathsf{flmap}(\lambda\ ((\mathsf{City}(k, l_1), \mathsf{City}(d, l_2)) \rightarrow (k, (d, l_2))), R)))))\end{array}$$

where R is a set of routes between two cities. Each city $\mathsf{City}(n, l)$ has a name n and a set of landmarks l and each landmark $\mathsf{Landmark}(n, r)$ has a rating r. $\mathsf{bestRated}(l_1, l_2)$ 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

$$\mu(S, \lambda\ (X \to \mathsf{flmap}(\lambda\ (x \to \mathsf{flmap}(\lambda\ (\mathrm{User}(u, bm) \to \mathsf{if}\ x \in bm\ \mathsf{then}\ bm\ \mathsf{else}\ \{\}), U)), X)))$$

where U is a set of users, each user $\mathrm{User}(u, bm)$ 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

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 $\bot$. It is defined as follows:

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

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

A lambda expression $f = \lambda\ (\pi_1 \to e_1 \mid \cdots \mid \pi_n \to e_n)$ 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 $\bot$. Let i be the smallest index such that $\mathrm{m}(v, \pi_i) = S \neq \bot$, 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: $H_f^{\otimes}(\{v_1\} \uplus \{v_2\} \uplus ... \uplus \{v_n\}) \rightsquigarrow f(v_1) \otimes f(v_2) \otimes ... \otimes f(v_n)$. As monoid operators are associative, parts of an expression in the form $e_1 \otimes e_2 \otimes ... \otimes e_n$ 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:

$$\frac{\varphi(R) \subset R}{\mu(R, \varphi) \rightsquigarrow \mathrm{distinct}(R)} \qquad \frac{\varphi(R) \not\subset R}{\mu(R, \varphi) \rightsquigarrow \mu(R \cup \varphi(R), \varphi)}$$

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:

```
res = R
new = R
while new ≠ ∅:
    new = φ(new)\res
    res = res ∪ new
return res
```

Distributed execution We consider in a distributed setting that distributed bags are partitionned. Distributed data is noted in the following way: $R = R_1|R_2|...|R_p$, 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:

- $H_f^{\uplus}(R_1|R_2|...|R_p) \rightsquigarrow H_f^{\uplus}(R_1)|H_f^{\uplus}(R_2)|...|H_f^{\uplus}(R_p)$ (partitionning does not have to change)
- $H_f^{\otimes}(R_1|R_2|...|R_p) \rightsquigarrow H_f^{\otimes}(R_1) \otimes_{nl} H_f^{\otimes}(R_2) \otimes_{nl} ... \otimes_{nl} H_f^{\otimes}(R_p)$, 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 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 operations like join 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:

- an optimization rule $\mathrm{P}_{\mathrm{dist}}$ that determines how a fixpoint term is evaluated in a distributed manner by choosing among two possible execution plans.
- 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. 3.2.

– PJ is a rewrite rule of the form:

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

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 premappability condition in Datatalog [26].

3.1 Distribution of the fixpoint operations ($\mathbf{P_{dist}}$)

As mentioned previously (Sec. 2.3), φ is a monoid homomorphism $\uplus \rightarrow \uplus$. This means that for $R = R_1 \uplus R_2$, $R \cup \varphi(R) = (R_1 \uplus R_2) \cup (\varphi(R_1) \uplus \varphi(R_2)) = R_1 \cup R_2 \cup \varphi(R_1) \cup \varphi(R_2)$.

One way to evaluate the distributed version of the fixpoint is:

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

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

Following this reduction rule, at each iteration, $(R_1 \cup \varphi(R_1)) \ \cup_{nl} \ (R_2 \cup \varphi(R_2)) \ \cup_{nl} ...$ will be evaluated, φ applied on the result and the stopping condition ($\varphi(R) \subset R$) checked. We name this execution plan $\mathcal{P}_1$. In the TC example (section 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:

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

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 $\mathrm{P_{dist}}$ uses the plan $\mathcal{P}_2$ instead of $\mathcal{P}_1$ for evaluating fixpoints.

3.2 Pushing filter inside a fixpoint (PF)

Filter depending on a single pattern variable

Definition 1 (filter). *We call* filter *a function of the form* $\lambda\ (D \rightarrow \mathit{flmap}(\lambda\ (\pi \rightarrow \mathit{if}\ c(a)\ \mathit{then}\ \{\pi\}\ \mathit{else}\ \{\}), D))$. *Such a function returns the dataset D filtered based on the condition c that depends on the variable a (where a appears in* π*).* π *is the pattern that matches each element of D and extracts from it the variable a that will be used to either keep the element or discard it.*

Let us consider a term $A = F(D)$ where F is such a filter. So we can say that given any $d \in D$, $d \in A \Leftrightarrow c(\pi_a(d))$, where $\pi_a(d)$ denotes the extraction of the pattern variable a from d by the pattern π. For instance, $\pi_a((1,(5,6))) = 5$ for $\pi = (x,(a,y))$.

Suppose now D is a fixpoint expression, i. e. $A \equiv \mathsf{flmap}(\lambda\ (\pi \rightarrow \mathsf{if}\ c(a)\ \mathsf{then}\ \{\pi\}\ \mathsf{else}\ \{\}), \mu(R,\varphi))$.

We want to explore sufficient conditions to push the filter before the fixpoint operation $\mu(R,\varphi)$. The goal is to find a generic way of analysing φ expressions regardless of which operations they contain and putting minimal restrictions on them that guarantee that the filter can be pushed safely.

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 C.4 [6]) that if (C) is satisfied, then the filter can be pushed, and the expression A would be equivalent to $\mu(R_f,\varphi)$, where $R_f = \mathsf{flmap}(\lambda\ (\pi \rightarrow \mathsf{if}\ c(a)\ \mathsf{then}\ \{\pi\}\ \mathsf{else}\ \{\}), R)$. In other words, the constant part R can be filtered first before applying the fixpoint on it.

This condition means that the operation φ does not change the value of a (the variable on which the filter depends), so each record of 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.

In order to verify that condition (C) is fulfilled, one way is to use the scala type inference system on the operation φ after translating it to a scala function. By giving this function a polymorphic input type and checking the output type, we could see where the variable a is in the output and whether it was changed.

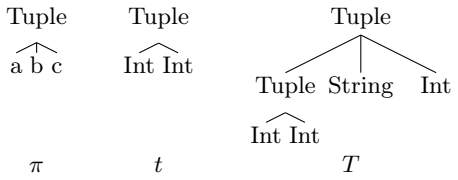
Verifying (C) using types We will start first by explaining the main idea behind this before going into the details. Types are made from type constructors and basic types, and patterns are made from type constructors and pattern variables. So we can represent their structures using trees. Since the type T of R elements matches the pattern π_a (otherwise the term would not be type correct, see Appendix A [6]), we expect their tree structures to match as well (we formalize this notion below using the 'correspondsTo' relation). So we can find in T the type t corresponding to the pattern variable a by following the structure of π (against the stucture of T). If we can affirm that this data of type t is not modified (or displaced) by the φ function, then the extraction of a by the pattern π from r and from $\varphi(\{r\})$ results would lead to the same value, hence (C).

For this, we use type checking and polymorphic types to follow the data of type t in the φ function. It relies on the fact that, 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.

Definition of the 'correspondsTo' relation Let T be a tree and n a node in this tree. We identify the path to n in T denoted path(n, T) by the sequence of pairs (number, label) that correspond to the ordered sequence of node label and next child arity for each node that is visited to reach n from the root of the tree.

Let T and T' be two trees. We define a relation in the set of T and T' nodes, denoted correspondsTo(T, T') by the following: Two nodes n_1 and n_2 are related (n_1 correspondsTo(T, T') n_2) when path(n_1, T) = path(n_2, T'). If two trees have the same structure, every node belonging to a tree will be related to a node in the other tree.

Let a be a pattern variable in π. Because π matches T, a correspondsTo(π, T) t for some node t in T. For example, in the figure below: a in the pattern π corresponds to the subtree t (or more precisely the root of the subtree) in the type T. In that case, we have a correspondsTo(π, T) t.



We consider the parametric type $C(t)$ obtained by replacing the mentionned subtree in T by the type parameter t. In the example, $C(t) = (t \times \mathit{String} \times \mathit{Int})$.

We have $\forall e : C(t) \quad \pi_a(e) : t$ and e: C($\{\pi_a(e)\}$), where $\{\pi_a(e)\}$ is the singleton type containing one element $\pi_a(e)$

We then translate the φ function to Scala, give it the polymorphic input type Bag$[C(t)]$ (with t at the position of the variable a on which the filter depends), and use the type inference system ([18]) to compute the output type. In case the function φ performs any specific operation (like addition or field access) on the element of type t other than copying it, a compilation error will be generated. Otherwise, the parameter t will appear in the output type. If the type inference system computes a return type Bag$[C'(t)]$, it means that for every arbitrary type t, φ can be given an argument of type Bag$[C(t)]$ and returns a value of type Bag$[C'(t)]$. So we have the following: $\forall r \in R\ r : C(\{\pi_a(r)\})$.
So, $\{r\}$: Bag$[C(\{\pi_a(r)\})]$. So, $\varphi(\{r\})$: Bag$[C'(\{\pi_a(r)\})]$. So, finally: $\forall s \in \varphi(\{r\})\ s : C'(\{\pi_a(r)\}$.

So if a correspondsTo(π, $C'(t)$) t, then $\pi_a(s) : \{\pi_a(r)\}$ meaning $\pi_a(r) = \pi_a(s)$.

Conclusion: In order for our filter pushing condition to be valid, it is sufficient for φ to have an output type Bag$[C'(t)]$ when given an input of type Bag$[C(t)]$ (with the type parameter t in $C(t)$ corresponds to a in π) and that t in $C'(t)$ corresponds to a in π. This means in other words that φ did not change the value

nor the position of a in the data it manipulates, so the filter can be performed on the data before applying φ multiple times, since this application will only generate a value with the same a for each of its input elements.

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

3.3 Filtering inside a fixpoint before a join (PJ)

Let us consider the expression: 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 only keeps 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 $join(A, B) = join(A, F_A(B))$, 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, that can be pushed when the criteria on pushing filters is fulfilled
3. we show as well that

$$\forall R, F_A(R) = \mathsf{flmap}(\lambda\ ((k, (s_v, s_w)) \rightarrow \mathsf{if}\ s_w \neq \{\}\ \mathsf{then}\ \mathsf{flmap}(\lambda\ (v \rightarrow \{(k, v)\}), s_v)\ \mathsf{else}\ \{\}), \mathsf{cogr}(R, A))$$

Proofs of the above are in appendix C.5 [6].

3.4 Pushing aggregation into a fixpoint (PA)

Let us consider a term of the form $f(\mu(R, \varphi))$. Assume that f is a homomorphism: $\cup \rightarrow \oplus$, and that f is idempotent. Typically, this is the case if f is a reduce or a reduceByKey.

Assume finally that we have $f \circ \varphi \circ f = f \circ \varphi$. Then 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 C.6 [6]).

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

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.

Rules PJ and $\mathrm{P}_{\mathrm{dist}}$ however require specific criteria to ensure that their application actually enhances performance.

Criteria for PJ The rule PJ introduces an additional cogroup to compute the filter being pushed in the fixpoint (as detailed in Sec. 3.3). 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 once only, 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 mentionned in (Sec. 2.5), all our algebraic operators apart from flatmap trigger non-local transfers. We then consider the following, where $\mathsf{size}_t(e)$ is the size of transfers incurred by the term e:

- We assume that $\mathsf{groupby}(A)$ 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, $\mathsf{size}_t(\mathsf{groupby}(A)) \approx o(\mathsf{size}(A))$
- Similarly, $\mathsf{cogr}(A,\ B)$ and $\mathsf{join}(A,\ B)$ transfer A and B (the cogroup and join operators are made between all elements of A and B):
  $\mathsf{size}_t(\mathsf{cogr}(A, B)) \approx o(\mathsf{size}(A) + \mathsf{size}(B))$
  $\mathsf{size}_t(join(A, B)) \approx o(\mathsf{size}(A) + \mathsf{size}(B))$
- $\mu(R, \varphi)$ would have to send all its result in order to compute the set union operation. So we just refer to $\mathsf{size}(\mu(R, \varphi))$ to indicate the size of the fixpoint result.

Let $S_1 = \mathsf{size}_t(join(A, \mu(R, \varphi)))$
and $S_2 = \mathsf{size}_t(join(A, \mu(F_A(R), \varphi)))$

$S_1 \approx o(\mathsf{size}(A) + 2 \times \mathsf{size}(\mu(R, \varphi)))$, 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.

$S_2 \approx o(2 \times \mathsf{size}(A) + 2 \times \mathsf{size}(\mu(F_A(R), \varphi)) + \mathsf{size}(R))$, 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: $2 \times \mathsf{size}(\mu(R, \varphi))$ and $2 \times \mathsf{size}(\mu(F_A(R), \varphi)) + \mathsf{size}(A) + \mathsf{size}(R)$. 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.

Criteria for PA The PA applies the function f on the fixpoint intermediate results. When f performs an aggregation (such as reduce or reduceByKey), the size of theses 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 semanticaly equivalent but the evaluation of the first does not terminate).

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.

Criteria for $\mathbf{P_{dist}}$ in the Spark setting The application of the rule $\mathrm{P}_{\mathrm{dist}}$ 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 $\varphi = join(X, S)$ 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 $\mathrm{P_{dist}}$ 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. 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 $\mathrm{PF}, \mathrm{PJ}, \mathrm{PA}, \mathrm{P_{dist}}$ (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 repeat queries in such a way that they compute the fixpoint more efficiently using the algorithm mentioned in 2.5. All DIQL queries used are given in Appendix D [6].
- **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 Rdist optimisation. We use it to assess the impact of the $\mathrm{P_{dist}}$ optimisation.

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 [13] dataset[3]), a social network graph (Facebook), and a scientific collaborations network (DBLP) taken from [15].

[3] We use a cleaned version of the real world dataset Yago 2s [13], that we have preprocessed in order to remove duplicate RDF triples 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).

– 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

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 [13]	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.

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)[4].

For the Yago dataset, transitive closures are computed for the `isLocatedIn` edge label. The hand written spark program (manual-spark) has the optimisations 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 4 presents the obtained results. We observe that the programs generated by mu-monoid systematically outperform the other program

[4] Each 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.

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}.

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 language (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 optimising programs. To enable automatic optimisations, 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 optimisations are not available for such constructs.

LINQ [17] and Ferry [12] are other comprehension based languages. However, unlike Emma, they do not analyse comprehensions to make optimisations. 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. 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. A complete survey of those works and of other EDSLs can be found in [3].

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 [23,19], monad comprehensions [24], ringad comprehensions [11].

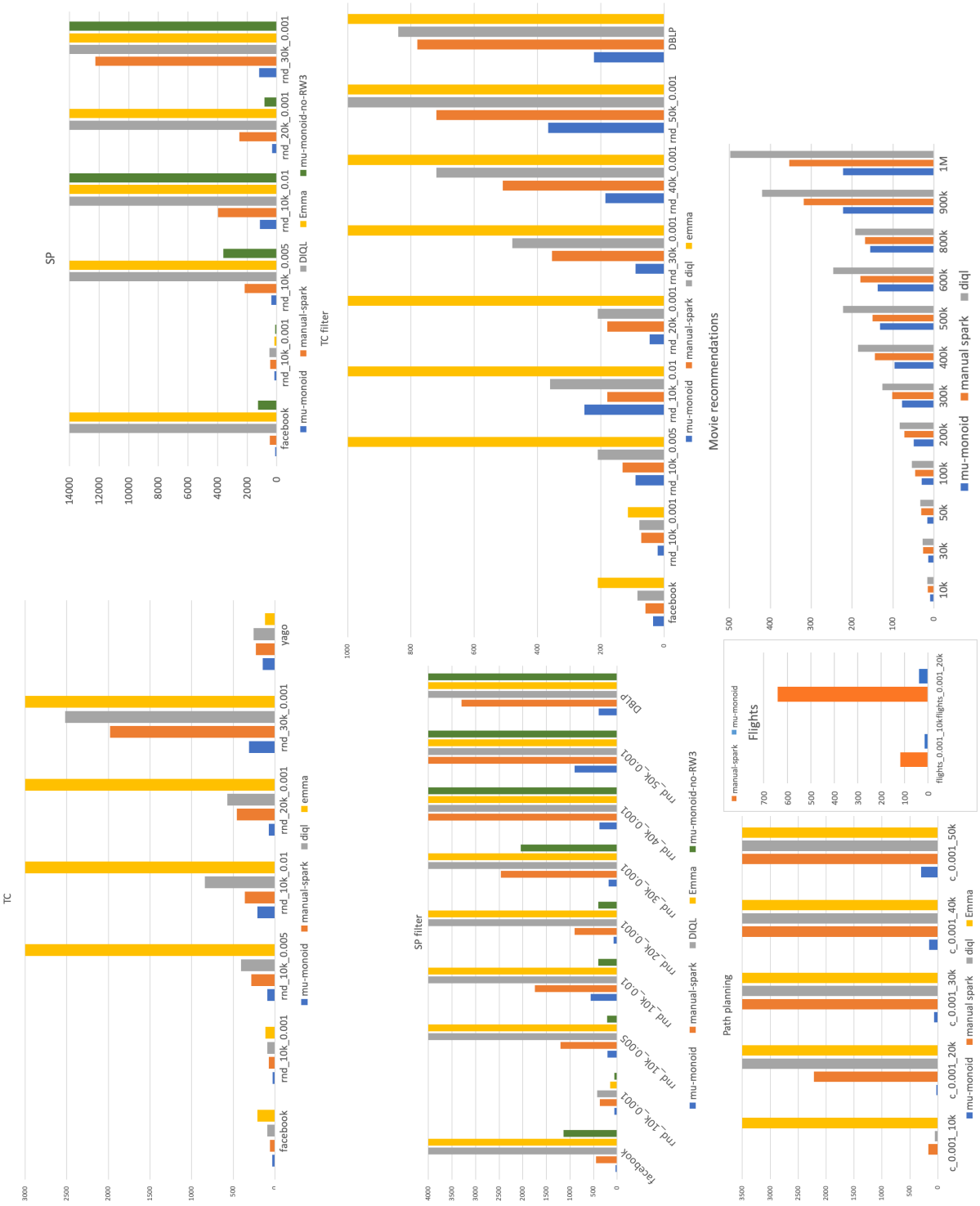


Fig. 4: Performance comparison of program versions. Program running times are reported in seconds (on the y-axes). A bar reaching the maximal value on the y-axis indicates a timeout.

The work found in [8] is pursuing a similar goal to that of Emma, which is to optimise 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 (Scala in this work). Using reflection of the host language and quotations, queries (of a DSL that translates to the algebra like DIQL [10]) 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 (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.

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 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.

References

1. Alexandrov, A., Katsifodimos, A., Krastev, G., Markl, V.: Implicit parallelism through deep language embedding. SIGMOD Record **45**(1), 51–58

(2016). https://doi.org/10.1145/2949741.2949754, https://doi.org/10.1145/2949741.2949754
2. Alexandrov, A., Krastev, G., Markl, V.: Representations and optimizations for embedded parallel dataflow languages. ACM Trans. Database Syst. **44**(1), 4:1–4:44 (Jan 2019). https://doi.org/10.1145/3281629, http://doi.acm.org/10.1145/3281629
3. Alexandrov, A., Krastev, G., Markl, V.: Representations and optimizations for embedded parallel dataflow languages. ACM Trans. Database Syst. **44**(1) (Jan 2019). https://doi.org/10.1145/3281629, https://doi.org/10.1145/3281629
4. Buneman, P., Libkin, L., Suciu, D., Tannen, V., Wong, L.: Comprehension syntax. SIGMOD Rec. **23**(1), 87–96 (Mar 1994). https://doi.org/10.1145/181550.181564, https://doi.org/10.1145/181550.181564
5. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink^{TM}: Stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**(4), 28–38 (2015), http://sites.computer.org/debull/A15dec/p28.pdf
6. Chlyah, S., Gesbert, N., Genevès, P., Layaïda, N.: On the Optimization of Iterative Programmingwith Distributed Data Collections (Oct 2020), https://hal.inria.fr/hal-02066649, full version of the present paper with appendix
7. Dean, J., Ghemawat, S.: 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. pp. 137–150 (2004), http://www.usenix.org/events/osdi04/tech/dean.html
8. Fegaras, L.: An algebra for distributed big data analytics. Journal of Functional Programming **27**, e27 (2017). https://doi.org/10.1017/S0956796817000193
9. Fegaras, L., Maier, D.: Optimizing object queries using an effective calculus. ACM Trans. Database Syst. **25**(4), 457–516 (2000), http://portal.acm.org/citation.cfm?id=377674.377676
10. Fegaras, L., Noor, M.H.: 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. pp. 1–8 (2018). https://doi.org/10.1109/BigDataCongress.2018.00008, https://doi.org/10.1109/BigDataCongress.2018.00008
11. Gibbons, J.: 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, pp. 132–151. Springer (2016). https://doi.org/10.1007/978-3-319-30936-1_7, https://doi.org/10.1007/978-3-319-30936-1_7
12. Grust, T., Mayr, M., Rittinger, J., Schreiber, T.: Ferry: Database-supported program execution. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data. p. 1063–1066. SIGMOD '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1559845.1559982, https://doi.org/10.1145/1559845.1559982
13. for Informatics, M.P.I., University, T.P.: YAGO: A high-quality knowledge base (july 2019), https://www.mpi-inf.mpg.de/yago-naga/yago/
14. Jachiet, L., Genevès, P., Gesbert, N., Layaïda, N.: On the optimization of recursive relational queries: Application to graph queries. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (2020 (to appear)), https://hal.inria.fr/hal-01673025v5/document, https://hal.inria.fr/hal-01673025v5/document
15. Leskovec, J.: Snap: Stanford large network dataset collection (november 2019), https://snap.stanford.edu/data/

16. Markl, V.: Emma is a quotation-based scala dsl for scalable data analysis. (november 2019), https://github.com/emmalanguage
17. Meijer, E., Beckman, B., Bierman, G.M.: 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. p. 706 (2006). https://doi.org/10.1145/1142473.1142552, https://doi.org/10.1145/1142473.1142552
18. Odersky, M., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M., et al.: An overview of the scala programming language. Tech. rep. (2004)
19. Peyton Jones, S.L.: The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1987)
20. Shkapsky, A., Yang, M., Interlandi, M., Chiu, H., Condie, T., Zaniolo, C.: Big data analytics with datalog queries on spark. In: Özcan, F., Koutrika, G., Madden, S. (eds.) Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016. pp. 1135–1149. ACM (2016). https://doi.org/10.1145/2882903.2915229, https://doi.org/10.1145/2882903.2915229
21. Tannen, V., Buneman, P., Naqvi, S.A.: 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. pp. 9–19 (1991)
22. Tannen, V., Buneman, P., Ohori, A.: Data structures and data types for object-oriented databases. IEEE Data Eng. Bull. **14**(2), 23–27 (1991), http://sites.computer.org/debull/91JUN-CD.pdf
23. Turner, D.A.: Recursion Equations as a Programming Language, pp. 459–478. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-30936-1_24, https://doi.org/10.1007/978-3-319-30936-1_24
24. Wadler, P.: Comprehending monads. Mathematical Structures in Computer Science **2**(4), 461–493 (1992). https://doi.org/10.1017/S0960129500001560, https://doi.org/10.1017/S0960129500001560
25. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016). https://doi.org/10.1145/2934664, http://doi.acm.org/10.1145/2934664
26. Zaniolo, C., Yang, M., Das, A., Shkapsky, A., Condie, T., Interlandi, M.: Fixpoint semantics and optimization of recursive datalog programs with aggregates. Theory Pract. Log. Program. **17**(5-6), 1048–1065 (2017). https://doi.org/10.1017/S1471068417000436, https://doi.org/10.1017/S1471068417000436

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

$$\frac{}{match(a, t) \to a : t}$$

$$\frac{\forall i \ match(\pi_i, t_i) \to \Gamma_i}{match(C(\pi_1, ..., \pi_n), C[t_1, ..., t_n]) \to \Gamma_1 \cup \cdots \cup \Gamma_n}$$

If, according to these rules, there is no Γ such that $match(\pi, t) \to \Gamma$ 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 3 (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:*

$$t + (t'_1 \mid\mid \cdots \mid\mid t'_m) = (t + t'_1) + (t'_2 \mid\mid \cdots \mid\mid t'_m)$$
$$t + C[t_1, ..., t_n] = t \mid\mid C[t_1, ..., t_n]$$
$$(t \mid\mid C[t_1, ..., t_n]) + C[t'_1, ..., t'_n] = t \mid\mid C[t_1 + t'_1, ..., t_n + t'_n]$$

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 $t + t' = t'$.

Definition 4 (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 5.* TODO: *put in figure: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 host language. An interesting particular case, however, is when a λ expression is used directly as

the first parameter of a $\mathsf{flmap}(,)$ or $\mathsf{reduce}(,)$. We can write the following compound deterministic rule in the case of $\mathsf{flmap}(,)$, for example:

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

$$\frac{\Gamma \vdash e_1 : t \to T_1[t'] \qquad \Gamma \vdash e_2 : T_2[t] \qquad T_1 = T_2 \vee (T_1 = \mathsf{Bag}_l \wedge T_2 = \mathsf{Bag}_d)}{\Gamma \vdash \mathsf{flmap}(e_1, e_2) : T_2[t']}$$

$$\frac{\Gamma \vdash e_1 : t \to t \to t \qquad \Gamma \vdash e_2 : T[t]}{\Gamma \vdash \mathsf{reduce}(e_1, e_2) : t} \qquad \frac{\Gamma \vdash e : T[t \times t']}{\Gamma \vdash \mathsf{groupby}(e) : T[t \times \mathsf{Bag}_l[t']]}$$

$$\frac{\Gamma \vdash e_1 : t' \to t' \to t' \qquad \Gamma \vdash e_2 : T[t \times t']}{\Gamma \vdash \mathsf{reduceByKey}(e_1, e_2) : T[t \times t']}$$

$$\frac{\Gamma \vdash e_1 : T_1[t \times t_1] \qquad \Gamma \vdash e_2 : T_2[t \times t_2] \qquad T_3 = (\mathsf{if}\ T_1 = T_2\ \mathsf{then}\ T_1\ \mathsf{else}\ \mathsf{Bag}_d)}{\Gamma \vdash \mathsf{cogr}(e_1, e_2) : T_3[t \times (\mathsf{Bag}_l[t_1] \times \mathsf{Bag}_l[t_2])]}$$

$$\frac{\Gamma \vdash e_1 : T_1[t \times t_1] \qquad \Gamma \vdash e_2 : T_2[t \times t_2] \qquad T_3 = (\mathsf{if}\ T_1 = T_2\ \mathsf{then}\ T_1\ \mathsf{else}\ \mathsf{Bag}_d)}{\Gamma \vdash \mathsf{join}(e_1, e_2) : T_3[t \times (t_1 \times t_2)]}$$

$$\frac{\Gamma \vdash e_1 : T[t] \qquad \Gamma \vdash e_2 : T[t] \to T[t]}{\Gamma \vdash \mu\,(e_1, e_2) : T[t]} \qquad \frac{\forall i\ \Gamma \vdash e_i : t_i}{\Gamma \vdash C(e_1, e_2, ..., e_n) : C[t_1, t_2, ..., t_n]}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \{e\} : \mathsf{Bag}_l[t]}$$

$$\frac{t'_1 + \cdots + t'_n = t' \qquad match(\pi_i, t'_i) \to \Gamma'_i \qquad \Gamma + \Gamma'_i \vdash e_i : t_i \qquad t_1 + \cdots + t_n = t}{\Gamma \vdash \lambda\ (\pi_1 \to e_1 \mid \cdots \mid \pi_n \to e_n) : t' \to t}$$

$$\frac{\Gamma \vdash e_1 : t_1 \to t' \qquad \Gamma \vdash e_2 : t_2 \qquad t_2 <: t_1}{\Gamma \vdash e_1\ e_2 : t'} \qquad \frac{\Gamma(a) = t}{\Gamma \vdash a : t} \qquad \frac{type(c) = t}{\Gamma \vdash c : t}$$

Fig. 5: Typing judgements.

B μ-monoids operators as monoid homomorphisms

flatmap operator We consider a function $f : \alpha \to \mathsf{Bag}[\beta]$. $\mathsf{flmap}(f, X)$ applies f on each element in the bag X and returns a dataset that is the union of all results. This operation is a monoid homorphism $H_f^{\uplus}$ from $(\mathsf{Bag}[\alpha], \uplus, \{\})$ to $(\mathsf{Bag}[\beta], \uplus, \{\})$.

reduce operator $\mathsf{reduce}(\oplus, X)$ reduces the elements of the input dataset by combining them with the $\oplus$ operator, which must be associative and commutative. This operation is a monoid homomorphism $H^{\oplus}_{id}$ from the monoid $\uplus$ to the monoid $\oplus$.

groupby operator groupby 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, $\mathsf{groupby}(\{(1,2),(1,4),(2,2),(2,1),(1,3)\}) = \{(1,\{2,4,3\}),(2,\{2,1\})\}$

groupby is a monoid homorphism $H^{\uparrow}_{f}$ from $(\mathsf{Bag}[\alpha \times \beta], \cup, \{\})$ to $(\mathsf{Bag}[\alpha \times \mathsf{Bag}[\beta]], \uparrow, \{\})$, where:

$$f\colon \alpha \times \beta \to \mathsf{Bag}[\alpha \times \mathsf{Bag}[\beta]]$$
$$(k, v) \mapsto \{(k, \{v\})\}$$

and the operator $\uparrow$ is defined such that

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

Note: Because of the way the function f is defined, the groupby operation supports bags of arbitrary types.

reduceByKey operator $\mathsf{reduceByKey}(\oplus, X)$ takes as X 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, which must be commutative and associative. The result is of the same type as X but each key appears exactly once in it.

If $\oplus$ operates on type β, then reduceByKey is a monoid homorphism $H^{\uparrow_\oplus}_{U_{\mathsf{Bag}}}$ from $(\mathsf{Bag}[\alpha \times \beta], \cup, \{\})$ to $(\mathsf{Bag}[\alpha \times \beta], \uparrow_\oplus, \{\})$, where the operator $\uparrow_\oplus$ is defined such that:

$$\{(k, b_1)\} \uparrow_\oplus \{(k', b_2)\} = \begin{cases} \{(k, b_1 \oplus b_2)\} & \text{if } k = k' \\ \{(k, b_1), (k', b_2)\} & \text{otherwise} \end{cases}$$

cogroup operator cogroup 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.

cogroup is a binary homomorphism $H^{\updownarrow}_{f_1, f_2}$ from the monoids $\uplus$ and $\uplus$ to the monoid $\updownarrow$, defined in the following way:

$$\mathsf{cogr}(X_1 \uplus Y_1, X_2 \uplus Y_2) = \mathsf{cogr}(X_1, Y_1) \updownarrow \mathsf{cogr}(X_2, Y_2)$$
$$\mathsf{cogr}(\{x\}, \{y\}) = f_1(x) \updownarrow f_2(y)$$

where:

$$f_1 \colon \alpha \times \beta \to \mathsf{Bag}[\alpha \times (\mathsf{Bag}[\beta]) \times \mathsf{Bag}[\gamma])]$$
$$(k, v) \mapsto \{(k, (\{v\}, \{\}))\}$$
$$f_2 \colon \alpha \times \gamma \to \mathsf{Bag}[\alpha \times (\mathsf{Bag}[\beta] \times \mathsf{Bag}[\gamma])]$$
$$(k, v) \mapsto \{(k, (\{\}, \{v\}))\}$$

and $\{(k, (b_1, c_1))\} \updownarrow \{(k', (b_2, c_2))\} =$

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

join operator join 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.

For a given A of type $\alpha \times \beta$, the operation $\mathsf{join}(A, \cdot)$ is a monoid homomorphism $H_f^{\uplus}$ where:

$$f \colon \alpha \times \gamma \to \mathsf{Bag}[\alpha \times (\beta \times \gamma)]$$
$$(k, v) \mapsto \{(k, (w, v)) \mid (k, w) \in A\}$$

Similarly, $\mathsf{join}(\cdot, A)$ is a monoid homomorphism.

C Additional Proofs

We have $\psi = R \uplus \varphi$ and we suppose (fc): $\forall A, B \quad \varphi(A \uplus B) = \varphi(A) \uplus \varphi(B)$

C.1 Proving the existence of the fixpoint

We prove that $\mu(\psi)$, the fixpoint of $f \colon X \to X \cup \psi(X)$ exists:

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

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

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

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 $F_1, ..., F_n$, all without duplicates such that $F = D_f \uplus F_1 \uplus ... \uplus F_n$ (they can be the singletons remaining after removing D_f from F)

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

So $f(F) = F \cup R \cup \varphi(D_f) \cup \varphi(F_1) \cup ... \cup \varphi(F_n)$

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 from bag union)

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 $\mu(\psi) = D_f = \text{distinct}(\biguplus_{n \in \mathbb{N}} \varphi^{(n)}(R)) = \bigcup_{n \in \mathbb{N}} \varphi^{(n)}(R)$

More properties of fixpoint expressions We have $\mu(R \uplus \varphi) = \bigcup_{n\in\mathbb{N}} \varphi^{(n)}(R) = \bigcup_{n\in\mathbb{N}} \varphi(\biguplus_{r\in R}\{r\}) = \bigcup_{n\in\mathbb{N}}(\biguplus_{r\in R}\varphi(\{r\})) = \bigcup_{n\in\mathbb{N}}(\bigcup_{r\in R}\varphi(\{r\}))$

Which means that $\forall a \in \mu(R \uplus \varphi)\ a \in \varphi^{(n)}(\{r\})$ for some $r \in R$ and $n \in N$
Also, $a \in \text{distinct}(\varphi^{(n)}(\{r\}))$ and $\forall n \in \mathbb{N}\ \forall r \in R\ \text{distinct}(\varphi^{(n)}(\{r\})) \subset \mu(\varphi)$

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

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

We put $F = \mu(R_1 \uplus R_2, \varphi)$

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

C.3 Bag formula for join

We have join(X,Y) = $\mathsf{flmap}(\lambda\ ((k,(s_x,s_y)) \to \mathsf{flmap}(\lambda\ (x \to \mathsf{flmap}(\lambda\ (y \to \{(k,(x,y))\}), s_y)), s_x)), \mathsf{cogr}(X,Y))$

We put $A = join(X,Y)$ and show that $A = B$, where $B = \{(k,(x,y)) \mid (k,x) \in X, (k,y) \in Y\}$

Following the bag definition of flatmap, we have $A = \biguplus_{(k,(s_x,s_y))\in\mathsf{cogr}(X,Y)} \biguplus_{x\in s_x} \biguplus_{y\in s_y} \{(k,(x,y))\}$

Let $e \in A$

$\exists (k,(s_x,s_y)) \in \mathsf{cogr}(X,Y)$ such that $\exists x \in s_x\ \exists y \in s_y \quad e = (k,(x,y))$

Since $(k,(s_x,s_y)) \in \mathsf{cogr}(X,Y)$, $s_x = \{v \mid (k,v) \in X\}$ (from the definition of cogroup), and since $x \in s_x$, then $(k,x) \in X$

Similarly, $(k,y) \in Y$

So $e = (k,(x,y)) \in B$

Let $(k,(x,y)) \in B$

We then have $(k,x) \in A$ and $(k,y) \in B$. So $k \in keys(A) \cup keys(B)$

Let $s_x = \{v \mid (k,x) \in X\}$ and $s_y = \{v \mid (k,y) \in Y\}$

So $(k,(s_x,s_y)) \in \mathsf{cogr}(A,B)$ and $x \in s_x$, $y \in s_y$, so $(k,(x,y)) \in A$

C.4 Condition for pushing a filter

Let

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

And let the condition (C) be

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

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

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, and can be shown using the following: $\forall s \in \mu(R, \varphi) \quad \exists r \in R \quad \exists n \in \mathbb{N} \quad s \in \varphi^{(n)}(\{r\})$ (shown in Appendix C.1).

1. $\mu(R_f, \varphi) \subset A$:
   $R_f \subset R \Rightarrow \mu(R_f, \varphi) \subset \mu(R, \varphi)$ (because $\mu(R, \varphi) = \mu(R_f \uplus R', \varphi) = \mu(R_f, \varphi) \cup \mu(R', \varphi)$ and $\mu(R, \varphi)$ does not contain duplicates)
   Let $s \in \mu(R_f, \varphi)$
   $\exists r \in R_f \quad \pi_a(s) = \pi_a(r)(*)$
   So $c(\pi_a(s)) = c(\pi_a(r))(=$ true because $r \in R_f)$
   So $s \in \mu(R, \varphi)$ and $c(\pi_a(s))$ is true, then $s \in A$
2. $A \subset \mu(R_f, \varphi)$:
   Let $s \in A, \quad s \in \mu(R, \varphi)$ and $c(\pi_a(s)) =$ true
   So $\exists r \in R \quad \exists n \in \mathbb{N} \quad \pi_a(s) = \pi_a(r)$ (*) and $s \in \varphi^{(n)}(\{r\})$
   So $c(\pi_a(r)) = c(\pi_a(s)) =$ true
   So $r \in R_f$, which means $\text{distinct}(\varphi^{(n)}(\{r\})) \subset \mu(R_f, \varphi)$ (see fixpoint related proofs in Appendix C.1), so $s \in \mu(R_f, \varphi)$

C.5 PJ proofs

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

So $(k, (x, y)) \in join(A, B) \Leftrightarrow (k, x) \in A \wedge (k, y) \in B \Leftrightarrow (k, x) \in A \wedge ((k, y) \in B \wedge \exists w \ (k, w) \in A) \Leftrightarrow (k, x) \in A \wedge (k, y) \in F(B) \Leftrightarrow (k, (x, y)) \in join(A, F(B)))$

Proving that F_A is a filter We can show that

$$\begin{aligned} F_A(B) &= \{(k, v) \mid (k, v) \in B, \exists w (k, w) \in A\} \\ &= \mathsf{flmap}(\lambda \ ((k, v) \rightarrow \mathsf{if} \ c(k) \ \mathsf{then} \ \{(k, v)\} \ \mathsf{else} \ \{\}), B) \end{aligned}$$

where $c(k)$ is the boolean expression that corresponds to the predicate $\exists w (k, w) \in A$

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

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

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

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

We have:

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

Let $e \in C$. So $\exists (k,(s_x,s_y) \in \mathsf{cogr}(B,A))$ such that $\exists x \in s_x \quad e = (k,x)$ and $s_y \neq \{\}$. We have $(k,(s_x,s_y) \in \mathsf{cogr}(B,A)$, 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 $e = (k,x) \in F_A(B)$.

Let $(k,x) \in F_A(B)$. We have $(k,x) \in B$ and $\exists w\ (k,w) \in A$. So $k \in keys(A) \cup keys(B)$. Let $s_x = \{v \mid (k,v) \in B\}$ and $s_y = \{v \mid (k,y) \in A\}$, so $(k,(s_x,s_y)) \in \mathsf{cogr}(B,A)$. Since $x \in s_x$ and $s_y \neq \{\}$ (because $\exists w\ (k,w) \in A$), then $(k,x) \in C$.

C.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 $\Phi\colon X \mapsto X \cup \varphi(X)$ and $\Psi\colon X \mapsto X \cup f \circ \varphi(X)$. 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:

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

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:

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

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

D DIQL Queries

Transitive closure:

```
REPEAT(X,O,oldCount, count) = (R,empty,d,c)
STEP (setDiff((SELECT DISTINCT (x, t)
    FROM (x,y) <- X, (z,t) <- R
    WHERE y == z),
    O),
    setUnion(O,X),
    count, O.count())
UNTIL count == oldCount
```

Shortest paths:

```
REPEAT (X,O,oldCount, count) = (R,empty,d,c)
STEP ((SELECT DISTINCT (x, t,l+m)
        FROM (x,y,l) <- X, (z,t,m) <- R
        WHERE y == z).coalesce(p).cache(),
      setUnion(O,X),
      count, O.count())
UNTIL count == oldCount
```

(reduceByKey is applied on the result of the query in the Scala program)

Path planning:

```
REPEAT(X,O,oldCount,count) = (source,empty,d,c)
STEP ((SELECT DISTINCT (x,cd.name,landmarkUnion(l,cd.landmarks) )
        FROM (x,y,l) <- X, (City(n1,l1), cd) <- routes
        WHERE y == n1).coalesce(p).cache(),
        setUnion(O,X),
        count, O.count())
UNTIL count == oldCount
```

Movie recommendations:

```
REPEAT(X,O,oldCount,count) = (startMovieNames,empty,d,c)
STEP (setDiff((SELECT DISTINCT m
         FROM (lm,e) <- (
           SELECT (lu, x) from User(u,lu) <- userslocal, x <- X
           WHERE lu.map(_.name).contains(x)), Movie(m) <- lm
         ),O),
       setUnion(O,X),
       count, O.count())
UNTIL count == oldCount
```