Representations and Optimizations for Embedded Parallel Dataflow Languages

ALEXANDER ALEXANDROV, GEORGI KRASTEV, and VOLKER MARKL, TU Berlin

Parallel dataflow engines such as Apache Hadoop, Apache Spark, and Apache Flink are an established alternative to relational databases for modern data analysis applications. A characteristic of these systems is a scalable programming model based on distributed collections and parallel transformations expressed by means of second oder functions such as `map` and `reduce`. Notable examples are Flink's `DataSet` and Spark's `RDD` programming abstractions. These programming models are realized as EDSLs – domain specific languages embedded in a general-purpose host language such as Java, Scala, or Python. This approach has several advantages over traditional external DSLs such as SQL or XQuery. First, syntactic constructs from the host language (e.g. anonymous functions syntax, value definitions, and fluent syntax via method chaining) can be reused in the EDSL. This eases the learning curve for developers already familiar with the host language. Second, it allows for seamless integration of library methods written in the host language via the function parameters passed to the parallel dataflow operators. This reduces the effort for developing analytics dataflows that go beyond pure SQL and require domain-specific logic.

At the same time, however, state-of-the-art parallel dataflow EDSLs exhibit a number of shortcomings. First, one of the main advantages of an external DSL such as SQL – the high-level, declarative `Select-From-Where` syntax – is either lost completely or mimicked in a non-standard way. Second, execution aspects such as caching, join order, and partial aggregation have to be decided by the programmer. Optimizing them automatically is very difficult due to the limited program context available in the intermediate representation of the DSL.

In this paper, we argue that the limitations listed above are a side effect of the adopted type-based embedding approach. As a solution, we propose an alternative EDSL design based on quotations. We present a DSL embedded in Scala and discuss its compiler pipeline, intermediate representation, and some of the enabled optimizations. We promote the algebraic type of bags in union representation as a model for distributed collections, and its associated structural recursion scheme and monad as a model for parallel collection processing. At the source code level, Scala's comprehension syntax over a bag monad can be used to encode `Select-From-Where` expressions in a standard way. At the intermediate representation level, maintaining comprehensions as a first-class citizen can be used to simplify the design and implementation of holistic dataflow optimizations that accommodate for nesting and control-flow. The proposed DSL design therefore reconciles the benefits of embedded parallel dataflow DSLs with the declarativity and optimization potential of external DSLs like SQL.

CCS Concepts: • **Software and its engineering → Parallel programming languages**; **Data flow languages**; *Functional languages*; • **Theory of computation →** *Algebraic semantics*; Categorical semantics;

Additional Key Words and Phrases: Parallel Dataflows; MapReduce; Monad Comprehensions

ACM Reference format:
Alexander Alexandrov, Georgi Krastev, and Volker Markl. 2018. Representations and Optimizations for Embedded Parallel Dataflow Languages. *ACM Trans. Datab. Syst.* 9, 4, Article 17 (March 2018), 52 pages.

This work has been supported through grants by the German Science Foundation (DFG) as FOR 1306 Stratosphere, by the German Ministry for Education and Research as Berlin Big Data Center BBDC (ref. 01IS14013A), by the European Commission as Proteus (ref. 687691) and Streamline (ref. 688191), and by Oracle Labs.
 0362-5915/2018/3-ART17 $15.00
DOI: 0000001.0000001

DOI: 0000001.0000001

1 INTRODUCTION

One of the key principles behind the pervasive success of data management technology and the emergence of a multi-billion dollar market in the past 40+ years is the idea of *declarative data processing*. The notion of *data* in this context has been traditionally associated with the relational model proposed by Codd [17]. The notion of *processing* has been traditionally associated with Relational Database Management Systems (RDBMSs). The notion of *declarativity* has two aspects: (*i*) the existence of high-level syntactic forms, and (*ii*) the ability to automatically optimize such syntactic forms by compiling them into efficient execution plans based on the relational algebra. Traditionally, (*i*) has been associated with the `Select-From-Where` syntax used in the Structured Query Language (SQL) [14], and (*ii*) with data-driven query compilation techniques [55]. Data management solutions based the *declarative data processing* paradigm therefore interface with clients through an external Domain Specific Language (DSL), most commonly SQL.

SQL is easy to teach and straight-forward to use for simple descriptive analytics, but is not so well-suited for more advanced analytics pipelines. The limitations of SQL are most evident in domains such as data integration or predictive data analysis. Programs in these domains are characterized by dataflows with features not directly supported by SQL, such as iterative computation, nested collections, and application-specific element-wise data transformations. To illustrate this, imagine a text processing pipeline that clusters text documents using an algorithm such as *k-means*. Conceptually, the input of such pipeline is a collection (a document corpus) of nested collections (the words for a specific document). The first part of the pipeline therefore has to operate on this nested structure in order to reduce each document into a suitable data point – for example a feature vector representing the *tf-idf* values of the words appearing in the document. The second part of the pipeline performs the actual clustering as a loop of repeated cluster re-assignment and centroid re-computation steps. Depending on the specific engine and SQL dialect, implementing this pipeline entirely in SQL ranges from impossible to cumbersome. If possible, an efficient encoding requires expert knowledge in advanced SQL features such as User-Defined Functions (UDFs) and User-Defined Types (UDTs) and control-flow primitives provided by a language extension such as PL/SQL. Technologies such as Microsoft's Language-Integrated Query (LINQ) mitigate some of these issues, but do not deal well with iterative dataflows.

In contrast, systems such as Apache Hadoop, Apache Spark, and Apache Flink offer a more flexible platform for data analysis pipelines. The notion of *processing* thereby corresponds to parallel dataflow engines designed to operate on very large shared-nothing clusters of commodity hardware. The notion of *data* corresponds to homogeneous distributed collections with user-defined element types. The notion of *declarativity*, however, is not mirrored at the language level. Instead, dataflow engines adopt a functional programming model where the programmer assembles dataflows by composing terms of higher-order functions, such as `map`(f) and `reduce`(g). The semantics of these higher-order functions guarantees a degree of data-parallelism unconstrained by the concrete function parameters (f and g). Rather than using a stand-alone syntax, the programming model is realized as a domain specific language embedded in a general-purpose host language, such as Java, Scala, or Python. This approach is more flexible, as it allows for seamless integration of data types and data processing functions from the host language ecosystem.

Despite this advantage, state-of-the-art Embedded Domain Specific Languages (EDSLs) offered by Spark (`RDD` and `Dataset`) and Flink (`DataSet` and `Table`) also exhibit some common problems. First, one of the main benefits of an external DSL such as SQL – the standardized declarative

`Select-From-Where` syntax – is either replaced in favor of a functional join-tree assembly or mimicked through function chaining in a non-standardized way. Second, execution aspects such as caching, join order, and partial aggregation are manually hard-coded by the programmer. Automatic optimization is either restricted or not possible due to the limited program context available in the Intermediate Representation (IR) constructed by the EDSL. As a consequence, in order to construct efficient dataflows, programmers must understand the execution semantics of the underlying dataflow engine. Further, hard-coding physical execution aspects in the application code increases its long-term maintenance cost and decreases portability.

In this paper, we argue that the problems listed above are a symptom of the type-based embedding approach adopted by these EDSLs. As a solution, we propose an alternative DSL design based on quotations. Our contributions are as follows:

- We analyze state-of-the-art EDSLs for parallel collection processing and identify their type-delimited nature as the root cause for a set of commonly exhibited deficiencies.
- As a solution to this problem, we propose *Emma* – a Scala DSL for parallel collection processing where DSL terms are delimited by *quotes* [52].[1] We discuss the concrete syntax and the Application Programming Interface (API) of *Emma*, its IR, and a compiler frontend that mediates between the two.
- We promote the algebraic type of bags in union representation as a model for distributed collections, and the associated structural recursion scheme (`fold`) and monad extension as a model for parallel collection processing.
- The formal model informs a systematic approach in the design of the *Emma* API and allows us to adopt some well-known database and language optimizations to the parallel dataflow domain. We also develop several new backend-agnostic and backend-specific optimizations that further demonstrate the utility of the proposed IR. Both the old and the new optimizations cannot be attained by the state-of-the art, type-delimited parallel dataflow EDSLs.
- We argue about the utility of monad comprehensions as a first-class syntactic form. At the source level, native comprehension syntax can be used to encode `Select-From-Where` expressions in a standard, host-language specific way, e.g., using for-comprehensions in Scala. At the IR level, treating comprehensions as a primitive form simplifies the definition and analysis of holistic dataflow optimizations in the presence of nesting and control-flow.
- We implement *Emma* backends that offload data-parallel computation on Apache Spark or Apache Flink, and demonstrate performance on par with hand-optimized code while attaining performance portability and lowering the requirements on the programmer.

The proposed design therefore can be seen as a step towards reconciling the flexibility of modern EDSLs for parallel collection processing with the declarativity and optimization potential of SQL.

The remainder of this paper is structured as follows. Section 2 reviews state-of-the-art technology and the research problem, while Section 3 outlines the proposed solution. Section 4 provides methodological background. Section 5 presents the abstract syntax and core API of *Emma*. Section 6 presents *Emma Core* – an IR suitable for optimization, and a transformation from *Emma Source* to *Emma Core*. Section 7.2 presents a translation scheme, similar to the one proposed by Grust [36], that

[1] In quote-delimited EDSLs, terms are not delimited by their type, but by an enclosing function call (called *quote*). For example, in the Scala expression

$$\texttt{onSpark}\,\{\,\texttt{for}\,\{\,\texttt{x} \leftarrow \texttt{xs}\,;\,\texttt{y} \leftarrow \texttt{ys}\,;\,\texttt{if}\,\texttt{k}_\texttt{x}(\texttt{x}) = \texttt{k}_\texttt{y}(\texttt{y})\,\}\,\texttt{yield}\,(\texttt{x}, \texttt{y})\,\}$$

the `onSpark` quote delimits the enclosed *Emma* code fragment. The `onSpark` implementation transforms the Abstract Syntax Tree of the quoted code into a program which evaluates the `for`-comprehension as a Spark dataflow with a `join` operator.

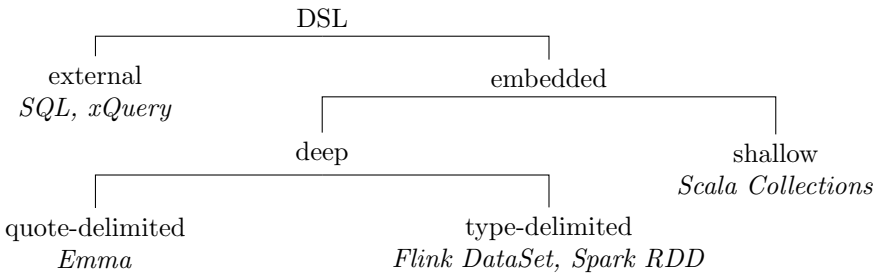


Fig. 1. Classification of DSLs. Examples in each class are given in *italic*.

maps *Emma Core* bag comprehensions to terms constructing parallel dataflow graphs. Section 7.3 shows how calls of the higher-order functions `map`, `filter`, and `join` in the resulting dataflow graphs can be further specialized as more efficient Spark `Dataset` operators. Based on well-known algebraic laws associated with the algebraic type of bags, Section 8 develops fusion-based optimizing transformations similar to the ones proposed in the context of functional programming [31, 59], ensuring that partial aggregates are introduced transparently as part of the *Emma* compilation process whenever possible. Section 9 presents a static analysis of *Emma Core* programs which allows us to identify and cache bag results consumed more than once. Section 10 illustrates another *Emma Core* analysis which is used to identify and specialize a restricted class of control-flow patterns as dedicated control-flow operators in Flink. Section 11 highlights the impact and importance of these optimizations through an experimental evaluation. Section 12 reviews related work, and, finally, Section 13 concludes and discusses future research.

2 STATE OF THE ART AND OPEN PROBLEMS

To motivate our work, we first introduce notions related to the implementation (Section 2.1) and design (Section 2.2) of DSLs relevant for the subsequent discussion. In Section 2.3, we then present a series of examples highlighting common problems with state-of-the-art parallel dataflow DSLs.

2.1 DSL Implementation Approaches

The DSL classes discussed below are depicted on Figure 1, with definitions adapted from [30]. With regard to their implementation approach and relation to General-purpose Programming Languages (GPLs), DSLs can be divided in two classes – *external* and *embedded*.

External DSLs define their own syntax and semantics. The benefit of this approach is the ability to define suitable language constructs and optimizations in order to maximize the convenience and productivity of the programmer. The downside is that, by necessity, external DSLs require a dedicated parser, type-checker, compiler or interpreter, tooling (e.g. for Integrated Development Environment (IDE) integration, debugging, and documentation), and possibly standard libraries. Examples of widely adopted external DSLs are SQL and Verilog.

Embedded Domain Specific Languages (EDSLs), first suggested by Hudak [38], are embedded into a GPL usually referred to as *host language*. Compared do external DSLs, EDSLs are more pragmatic to develop, as they can reuse the syntax, tooling, and third-party libraries of their host language.

Based on the embedding strategy, EDSLs can be further differentiated into two sub-classes. With a *shallow embedding*, DSL terms are implemented *directly* by defining their semantics as

host language expressions. With a *deep embedding*, DSL terms are implemented *reflectively* by constructing an IR of themselves. The IR then is optimized and either interpreted or compiled.[2]

Finally, the method used to delimit EDSL terms in host language code yields two more subclasses. With the *type-based* approach, the EDSL consists purely of a collection of GPL types, and the operations on these types are defined to construct the associated EDSL IR. Host language terms that belong to the EDSL are thereby delimited by their type. With the *quote-based* approach, the EDSL derives its IR from a host-language Abstract Syntax Tree (AST) using the reflection capabilities of the host language. EDSL terms are thereby delimited by the surrounding quotation.

2.2 EDSL Design Objectives

In order to improve the learning curve and adoption of EDSLs, their design is guided by three main principles. The first principle is to *maximize syntactic reuse* – that is, exploit the programmer's familiarity with syntactic conventions and tools from the host language and adopt those as part of the EDSL. The second principle is to *minimize syntactic noise* – that is, reduce the amount of idiosyncratic constructs specific to the EDSL. Adhering to the first two principles ensures that developers that are already familiar with the host language can start writing new or maintain existing DSL programs with minimal learning effort. In the case of parallel dataflow DSLs discussed in this paper, this means that host-language features, such as lambda functions, `for`-comprehensions, and control-flow statements, are part of the DSLs syntax. The third principle is to *simultaneously maximize program performance* through automated domain-specific optimizations. In the case of parallel dataflow DSLs, this means that program aspects related to execution, such as join order, intermediate result caching, as well as use of partial aggregates or dedicated control-flow runtime operators, are hidden from the programmer and introduced transparently by the DSL compiler. Next, we illustrate how state-of-the-art parallel dataflow DSLs violate these design principles.

2.3 Parallel Dataflow DSLs

2.3.1 Spark RDD and Flink DataSet. Early systems for web-scale data management, such as MapReduce [19] and Pregel [48], allowed users to process data flexibly and at a scale that was not possible with RDBMSs. However, encoding arbitrary dataflows in the fixed shapes offered by those systems was cumbersome to program, hard to optimize, and inefficient to execute. Next-generation dataflow engines and programming models, such as Spark [68] and Nephele/PACTs [7] (which evolved into Stratosphere/Flink), were designed to address these limitations.

Generalizing MapReduce, these systems were able to execute dataflow graphs composed freely from a base set of second-order operators. Going beyond `map` and `reduce`, this set was extended with binary operators such as `join`, `coGroup` and `cross`. To construct a dataflow graph in a convenient way, the systems offer type-based DSLs deeply embedded in JVM-based GPLs like Scala or Java. The core construct of both EDSLs is a generic type representing a distributed, unordered collection of homogeneous elements with duplicates. This type is called `RDD` (short for *Resilient Distributed Dataset*) in Spark, and `DataSet` in Stratosphere/Flink.

Compared to Hadoop's MapReduce APIs, the `RDD` and `DataSet` EDSLs significantly improve the assembly of dataflows. However, a closer look reveals a number of shared limitations. To illustrate

[2] Traditionally, shallow EDSLs are considered more intuitive, because one can reuse the entire host language syntax in the DSL, while deep EDSLs are considered to offer better performance, because one can analyze and optimize the reflected IR. In this paper, we adopt a method (quotations), which allows us to partially overcome the limitations of deep EDSLs and reuse more of the host language syntax. It should be noted, however, that recent research [46] demonstrates that one can also go the opposite route and partially overcome the limitations of shallow EDSLs using online partial evaluation.

those, we use a series of examples based on a simplified film database schema[3].

$$\begin{aligned} \texttt{Person} &= (\texttt{id} : \texttt{Long}) \times (\texttt{name} : \texttt{String}) \\ \texttt{Credit} &= (\texttt{personID} : \texttt{Long}) \times (\texttt{movieID} : \texttt{String}) \times (\texttt{creditType} : \texttt{String}) \\ \texttt{Movie} &= (\texttt{id} : \texttt{Long}) \times (\texttt{title} : \texttt{String}) \times (\texttt{year} : \texttt{Short}) \times (\texttt{titleType} : \texttt{String}) \end{aligned}$$

Example 2.1 (Operator Chains). To showcase the similarity between the RDD and DataSet EDSLs, consider a Scala code that filters movies from the 1990s and projects their year and name. Modulo the underlying collection type, the code is identical (the color coding will be explained later).

```
val titles = movies // either RDD[Movie] or DataSet[Movie]
  .filter( m => m.year >= 1900 ) // (1)
  .map( m => (m.year, m.title) ) // (2)
  .filter( m => m._1 < 2000 )    // (3)
```

Executing this code in Scala will append a chain of `filter` (1), a `map` (2), and a `filter` (3) operators to the dataflow graph referenced by `movies` and reference the resulting graph from a new RDD/DataSet instance bound to `titles`. This functional, fluent style of dataflow assembly is concise and elegant, but not really declarative and hard to optimize. To illustrate why, compare the code above with the equivalent SQL statement.

```
CREATE VIEW titles AS
SELECT m.year, m.title FROM movies AS m WHERE m.year >= 1900 AND m.year < 2000
```

A SQL optimizer will push the two selection predicates behind the projection. In the RDD/DataSet dataflow graphs, however, swapping (2) and (3) implies also adapting the function passed to (3), as the element type changes from (Short, String) to Movie. Since the IRs of both EDSLs treat functions bound to higher-order operators as black-box values, this rewrite cannot be realized directly. To implement those, one has to resort to bytecode analysis and manipulation [39].

Note that Scala's for-comprehensions offer a host language construct syntactically equivalent to SQL's Select-From-Where, so in principle the SQL compilation strategy outlined above can be applied on top of for-comprehensions. However, neither Flink nor Spark supports this currently.

Example 2.2 (Join Cascades). For this example, consider the following code fragments that relate movies with people based on the available credits[4].

```
// RDD (Spark)
val xs = movies.keyBy(_.id)
  .join(credits.keyBy(_.movieID)).values
val ys = xs.keyBy(_._2.personID)
  .join(people.keyBy(_.id)).values
```

```
// DataSet (Flink)
val xs = (movies join credits)
  .where(_.id).equalTo(_.movieID)
val ys = (xs join people)
  .where(_._2.personID).equalTo(_.id)
```

Two problems become evident from the above snippets. First, a standard, declarative syntax like *Select-From-Where* in SQL is not available in the RDD and DataSet EDSLs. Instead, *n*-ary joins have to be specified as cascades of binary `join` operators. The elements in the resulting collections are tuples of nested pairs whose shape mirrors the producing join tree. Subsequent field access therefore require projection chains that traverse the nested tuple tree to its leafs. For example, the type of `ys` is ((Movie, Credit), Person), and projecting (movie title, person name) pairs from `ys` can be done in one of two ways.

[3] Product (or *struct*) types can be encoded as case classes in Scala and used as a data model in both EDSLs.
[4] The following examples use Scala's short-hand function declaration syntax. For example, the term `_.id` denotes a function where _ is a parameter placeholder, i.e., it is semantically equivalent to `x => x.id`.

```
// total function with field projections
ys.map(y => {
  val m = y._1._1; val p = y._2
  (m.title, p.name)
})
```

```
// partial function with pattern matching
ys.map {
  case ((m, c), p) =>
    (m.title, p.name)
}
```

The second problem again is related to the ability to optimize constructed IR terms. Consider a situation where the code listed above represents the entire dataflow. Since not all base data fields are actually used, performance can be improved through insertion of early projections. In addition to that, changing the join order might also be beneficial. For the same reason stated in Example 2.1 (black-box function parameters), neither of these optimizations is possible in the discussed EDSLs. Current solutions indicate the potential benefits of such optimizations, but either depart from the syntactic reuse principle [6, 44] or rely on an auxiliary bytecode inspection or bytecode decompilation step [37, 39]. As in the previous example, a design based on `for`-comprehensions seems like a natural fit.

Example 2.3 (Reducers). Computing global or per-group aggregates is an integral operation in most analytics pipelines. This is how we can get the total number of movies using `map` and `reduce`.

```
movies // either RDD[Movie] or DataSet[Movie]
  .map(_ => 1L)
  .reduce((u, v) => u + v)
```

And this is how we can get the number of movies per decade.

```
// RDD (Spark)
movies
  .map(m => (decade(m.year), 1L))
  .reduceByKey((u, v) => u + v)
```

```
// DataSet (Flink)
movies
  .map(m => (decade(m.year), 1L))
  .groupBy(_._1)
  .reduce((u, v) => (u._1, u._2 + v._2))
```

The `reduce` and `reduceByKey` operators enforce an execution strategy where the input values (for each group) are reduced to a single aggregate value (per group) in parallel. This is achieved by means of repeated application of an associative and commutative binary function specified by the programmer and passed to the `reduce`/`reduceByKey` operators. Aggressive use of reducers therefore is essential for dataflow performance and scalability.

Nevertheless, optimal usage patterns can be hard to identify, especially without a good background in functional programming. For example, to check who between Alfred Hitchcock or Woody Allen has directed more movies, one might build upon the `ys` collection from Example 2.2.

```
val c1 = ys // count movies directed by Alfred Hitchcock
  .filter(_._1._2.creditType == "director").filter(_._2.name == "Hitchcock, Alfred")
  .map(_ => 1L).reduce((u, v) => u + v)
val c2 = ys // count movies directed by Woody Allen
  .filter(_._1._2.creditType == "director").filter(_._2.name == "Allen, Woody")
  .map(_ => 1L).reduce((u, v) => u + v)
c1 < c2 // compare the two counts
```

One problem with this specification is that it requires two passes over `ys`. A skilled programmer will achieve the same result in a single pass.

```
val (c1, c2) = ys // pair-count movies directed by (Alfred Hitchcock, Woody Allen)
```

```
  .filter(_._1._2.creditType == "director")
  .map(y => (
    if (y._2.name == "Hitchcock, Alfred") 1L else 0L,
    if (y._2.name == "Allen, Woody") 1L else 0L
  )).reduce((u, v) => (u._1 + v._1, u._2 + v._2))
c1 < c2 // compare the two counts
```

A second pitfall arises when handling groups. As group values cannot always be processed by associative and commutative functions, the discussed EDSLs provide means for holistic group processing with one UDF call per group. We can also count the movies per decade as follows.

```
// RDD (Spark)
movies
  .groupBy(m => decade(m.year))
  .map { case (k, vs) => {

    val v = vs.size
    (k, v)
  }}
```

```
// DataSet (Flink)
movies
  .groupBy(m => decade(m.year))
  .reduceGroup(vs => {
    val k = decade(vs.next().year)
    val v = 1 + vs.size
    (k, v)
  })
```

A common mistake is to encode a dataflow in this style even if it can be defined with a `reduce` operator. The above approach requires a full data shuffle, while in the `reduce`-based variants the size of the shuffled data is reduced by pushing some `reduce` computations before the shuffle step.

As with the previous two examples, optimizing these cases through automatic term rewriting is not possible in the `RDD` and `DataSet` EDSLs. Instead, constructing efficient dataflows is predicated on the programmer's understanding of the operational semantics of `reduce`-like operators.

Example 2.4 (Caching). Dataflow graphs constructed by `RDD` and `DataSet` terms are sometimes related by the enclosing host language program. For example, in the naïve "compare movie-counts" implementation from Example 2.3 the `ys` collection is referenced twice – once when counting the movies for Hitchcock (`c1`) and once for Allen (`c2`). Since a global `reduce` implicitly triggers evaluation, the dataflow graph identified by `ys` is also evaluated twice. To amortize the evaluation cost of the shared subgraph, the `RDD` EDSL offers a dedicated `cache` operator (in Flink, `cache` can be simulated by a pair of `write` and `read` operators).

```
val us = ys // cache the shared subgraph
  .filter(_._1._2.creditType == "director")
  .cache()
val c1 = us // count movies directed by Alfred Hitchcock
  .filter(_._2.name == "Hitchcock, Alfred")
  .map(_ => 1L).reduce((u, v) => u + v)
val c2 = us // count movies directed by Woody Allen
  .filter(_._2.name == "Allen, Woody")
  .map(_ => 1L).reduce((u, v) => u + v)
c1 < c2 // compare the two counts
```

Data caching also can significantly improve performance in the presence of control-flow, which is often the case in data analysis applications. To demonstrate this, consider a scenario where a collection `w` representing the parameters of some Machine Learning (ML) model is initialized and subsequently updated N times with the help of a static collection `S`.

```
// RDD (Spark)
val S = static().cache()
var w = init()
for (i <- 0 until N) {
  w = update(S, w).cache()
}
```

```
// DataSet (Flink)
val S = static()
var w = init()
w = w.iterate(N) ( w =>
  update(S, w)
)
```

The Spark version requires two explicit `cache` calls. If we do not cache the `static()` result, the associated dataflow graph will be evaluated N times. If we do not cache the `update()` result, the loop body will be replicated N times without enforcing evaluation. The Flink version automatically caches loop-invariant dataflows. In order to do this, however, the `DataSet` EDSL requires a dedicated `iterate` operator which models a restricted class of control-flow structures – a violation of the *maximize syntactic reuse* design principle.

2.3.2 Current Solutions. Two solution approaches are currently pursued in order to address the above problems. The first is to use an external DSLs. Notable external DSLs in this category are Pig Latin [53] and Hive [61]. This recovers both the declarative syntax and the advanced optimizations, as the entire AST of the input program can be reflected by the external DSL compiler. Unfortunately, it also brings back the original problems associated with SQL – lack of flexibility and treatment of UDFs and UDTs as second-class constructs.

The second approach is to promote expressions passed to dataflow operators such as `filter`, `select` and `groupBy` from "black-box" host-language lambdas to inspectable elements in the EDSL IR. Notable examples in this category are `DataFrame` and `Dataset` EDSLs in Spark [6] and the `Table` EDSL in Flink [44]. This enables logical optimizations such as join reordering, filter and selection push-down, and automatic use of partial aggregates. The problem is that one loses the ability to reuse host-language syntax (e.g. field access, arithmetic operators) in the first-class expression language. Instead, expressions are modeled either by plain old strings or by a dedicated type (`Expression` in Flink, `Column` in Spark). Syntactic reuse is violated in both cases. The following code illustrates the two variants in the Spark `DataFrame` (left) and the Flink `Table` (right) EDSLs.

```
credits.toDF() // string-based
  .select("creditType", "personID")
  .filter("creditType == 'director")
creditsDf.toDF() // type-based
  .select($"creditType", $"personID")
  .filter($"credytType" === "director")
```

```
credits.toTable(tenv) // string-based
  .select("creditType, personID")
  .where("creditType == 'director")
credits.toTable(tenv) // type-based
  .select(`creditType, `personID)
  .where(`credytType === "director")
```

Neither of the two variants benefits from the type-safety or syntax checking capabilities of the host language. For example, the `filter` expression in the string-based variant is syntactically incorrect, as it lacks the closing quote after `director`, and in the type-based variants the last `creditType` is misspelled. The enclosing Scala programs, however, will compile without a problem. The errors will be caught only at runtime, once the EDSL attempts to evaluate the resulting dataflow. In situations where long-running, possibly iterative computations are aborted at the very end due to a syntactic error, these issues can be particularly frustrating. Adding insult to injury, `filter` is overloaded to accept black-box Scala lambdas next to the more restricted but powerful DSL-specific expressions. The burden of navigating these alternatives once again is on the programmer.

3 QUOTE-DELIMITED PARALLEL DATAFLOW EDSLS

Section 2.3 outlined a number of limitations shared between the `DataSet` and `RDD` EDSLs and problems with current solutions. To find the root cause of these issues we position these EDSLs in the design space outlined in Section 2.2. Observe that in both systems, EDSL terms are delimited by their type. Because of this, the EDSL IR can only reflect method calls on these types and their def-use relation. The code fragments reflected in the IR in Section 2.3 were printed in **`bold teletype`** font. The remaining syntax (printed in `regular teletype` font) could not be represented in the IR. Notably, this encompasses the control-flow instructions and lambdas passed as operator arguments.

Type-delimited EDSLs suffer from restricted optimization and syntactic reuse potential. Optimizations such as operator reordering (Example 2.1), join-order optimization, insertion of partial aggregates (Example 2.3), and selection of caching strategies (Example 2.4, Spark) cannot be automated. In addition, syntactic forms such as `for`-comprehensions (Example 2.2) and control-flow primitives (Example 2.4, Flink) that might be a natural fit are not reused in the EDSL syntax.

We end up with EDSLs that seem straight-forward to use, yet for most applications require expert knowledge in data management and distributed systems in order to produce fast and scalable programs. The benefits of a declarative, yet performant language such as SQL are lost.

As a solution for this problem, we propose a quote-delimited DSL for parallel collection processing embedded in Scala. Quote-delimited EDSLs allow for deeper integration with the host language and better syntactic reuse. In addition, a more principled design of the collection processing API and the IRs of our EDSL enables the optimizations outlined above. The result is a language where notions of data-parallel computation no longer leak to the programmer. Instead, parallelism becomes implicit for the programmer without incurring significant performance penalty.

4 METHODOLOGY

This section gives methodological background relevant to our approach. Section 4.1 outlines an algebraic foundation for distributed collections and parallel collection processing based on Algebraic Data Types (ADTs), structural recursion, and monads. Section 4.2 reviews Static Single Assignment (SSA) form and a functional encoding of SSA called Administrative Normal Form (ANF).

4.1 Algebraic Foundations

4.1.1 Algebraic Data Types. We use ADTs to capture the essence of the distributed collection types we want to target. In this approach, the set of elements of a type T is defined as the least fixpoint of all terms that can be inductively constructed from a set of primitive functions [45]. For example, the type of natural numbers can be defined as an ADT as

$$\mathbb{N} = 0 \mid \mathbb{N} + 1 \ .$$

The right-hand-side of the equation defines ways to construct natural numbers (i) $0 \in \mathbb{N}$, (ii) if $x \in \mathbb{N}$ then $x + 1 \in \mathbb{N}$. The equals sign states that every $x \in \mathbb{N}$ can be uniquely encoded as a finite term over the 0 and $\cdot + 1$ constructors.

Collection types can be defined as ADTs in a similar way. For example, we can define the polymorphic type of homogeneous lists with elements of type A as follows.

$$\texttt{List}[A] = \mathit{emp} \mid \mathit{cons}\, A\, \texttt{List}[A] \qquad \text{(List-Ins)}$$

The *emp* constructor denotes the empty list, while $\mathit{cons}(x, xs)$ denotes the list constructed by inserting the element x in the beginning of a list xs. The same constructors can be used to define the collection types $\texttt{Bag}[A]$ and $\texttt{Set}[A]$ as ADTs. To match the semantics of each type, we constrain the definitions with suitable axioms, giving rise to the so-called Boom hierarchy of types [10]. To

generalize lists to bags, we introduce an axiom stating that order of insertion is not relevant.

$$cons(x, cons(x', xs)) = cons(x', cons(x, xs))$$

To generalize bags to sets, we introduce an axiom stating that element insertion is idempotent.

$$cons(x, cons(x, xs)) = cons(x, xs)$$

Collection types in Spark and Flink do not guarantee element order and allow for duplicates, so they are most accurately modeled as bags. As $\mathsf{Bag}[A]$ is the type underlying the formal foundation for our approach, the rest of the discussion in this section is focused on this type.

Depending on the choice of constructors, the Boom hierarchy can be defined in two ways. The definition used above is known as *insert representation*, as it models element insertion as a primitive constructor. Alternatively, one can use the *union representation*, which for bags looks as follows.

$$\mathsf{Bag}[A] = emp \mid sng\, A \mid uni\, \mathsf{Bag}[A]\, \mathsf{Bag}[A] \qquad \text{(Bag-Union)}$$

Here, *emp* denotes the empty bag, $sng(x)$ denotes a bag consisting only of x, and $uni(xs, ys)$ denotes the union of two bags. The intended bag semantics are imposed by the following axioms.

$$uni(xs, emp) = uni(emp, xs) = xs \qquad \text{(Bag-Unit)}$$

$$uni(xs, uni(ys, zs)) = uni(uni(xs, ys), zs) \qquad \text{(Bag-Assc)}$$

$$uni(xs, ys) = uni(ys, xs) \qquad \text{(Bag-Comm)}$$

Bag-Unit states that *emp* is neutral with respect to *uni*, Bag-Assc that *uni* is associative (i.e., evaluation order is irrelevant), and Bag-Assc that *uni* is commutative (i.e., element order is irrelevant).

Distributed collections in Spark and Flink are partitioned across different nodes in a shared-nothing cluster. The value of a distributed collection is defined as the disjoint union of all its partitions: $xs = \biguplus_{i=1}^{n} xs_i$. The union representation provides a model that can express this definition directly, and is therefore preferred in our work in favor of the insert representation, which is better known as it is more commonly used in functional programming textbooks.

4.1.2 Structural Recursion. Every ADT comes equipped with a structural recursion scheme called *fold*. The *fold* operator is (i) polymorphic in its return type B and (ii) parameterized by functions corresponding to the constructors of the associated ADT. For example, the *fold* operator for $\mathsf{Bag}[A]$ is parameterized by the functions $zero : B$, $init : \mathsf{Bag}[A] \Rightarrow B$, and $plus : B \times B \Rightarrow B$.[5] Partially applying *fold* with concrete instances of *zero*, *init* and *plus* yields a function $f : \mathsf{Bag}[A] \Rightarrow B$ which operates in three steps. First, it recursively parses the constructor application tree of the corresponding ADT value. Second, it substitutes constructor calls with corresponding function calls. Finally, the resulting tree is evaluated in order produce a final result of type B. Formally, this process can be defined as follows.

$$fold(zero, init, plus)(xs) = f(xs) = \begin{cases} zero & \text{if } xs = emp \\ init(x) & \text{if } xs = sng(x) \\ plus(f(us), f(vs)) & \text{if } xs = uni(us, vs) \end{cases} \qquad \text{(Bag-Fold)}$$

In order to ensure that the partial application $fold(zero, init, plus) = f$ is a well-defined function, the *zero*, *init*, and *plus* functions must satisfy the same bag axioms as the corresponding bag constructors *emp*, *sng*, and *uni*.[6]

[5] The type constructor $A \Rightarrow B$ denotes the type of functions mapping arguments of type A to results of type B, and the type constructor $A \times B$ denotes the type of tuples whose first first and second elements respectively have type A and B.

[6] In other words, the triple (B, *zero*, *plus*) must form a commutative monoid over B. In this perspective, f can be seen as a homomorphism in the category of commutative monoids. Specifically, $f = hom(init)$ is the unique extension of *init*

Structural recursion offers a semantically restricted, yet expressive model that captures the essence of parallel collection processing. The key insight is that the algebraic properties of the Bag ADT ensure parallel execution regardless of the concrete *zero*, *init*, and *plus* functions. For a distributed collection *xs*, this means that we can employ function shipping and evaluate $f(xs_i)$ on each partition before computing the final result using *plus*. In the functional programming community, this idea was highlighted by Steele [43]. In the Flink and Spark communities, the underlying mathematical principles seem to be largely unknown, although projects like Summingbird [11] and MRQL [23] demonstrate the benefits of bridging the gap between theory and practice. In addition, the fundamental relevance of *fold* is indicated by the fact that *fold* variations (under different names), as well as derived operators (such as *reduce*) are an integral part of the Flink and Spark APIs.

4.1.3 Monads & Monad Comprehensions. The Bag ADT can be extended to an algebraic structure known as *monad with zero*. A monad with zero is a tuple (*emp*, *sng*, *map*, *flatten*) where the first two functions coincide with the Bag constructors, and the last two can be defined as *fold* instances.

$$\begin{aligned} map(f : A \Rightarrow B) : \texttt{Bag}[A] \Rightarrow \texttt{Bag}[B] &= fold(emp, sng \circ f, uni) \\ flatten : \texttt{Bag}[\texttt{Bag}[A]] \Rightarrow \texttt{Bag}[B] &= fold(emp, id, uni) \end{aligned} \tag{Bag-Monad}$$

We refer the reader to Wadler [66] for a comprehensive introduction to monads. A monad permits a declarative syntax known as *monad comprehensions*. For the Bag ADT, the syntax is in one-to-one correspondence with the Select-From-Where syntax and semantics known from SQL. For example, a SQL expression that computes an equi-join between *xs* and *ys*

$$\textbf{SELECT } x.b,\ y.d \textbf{ FROM } xs \textbf{ as } x,\ ys \textbf{ as } y \textbf{ WHERE } x.a = y.c$$

corresponds to the following Bag comprehension (given in abstract syntax).

$$[\ x.b,\ y.d \mid x \leftarrow xs,\ y \leftarrow ys,\ x.a = y.c\ ]$$

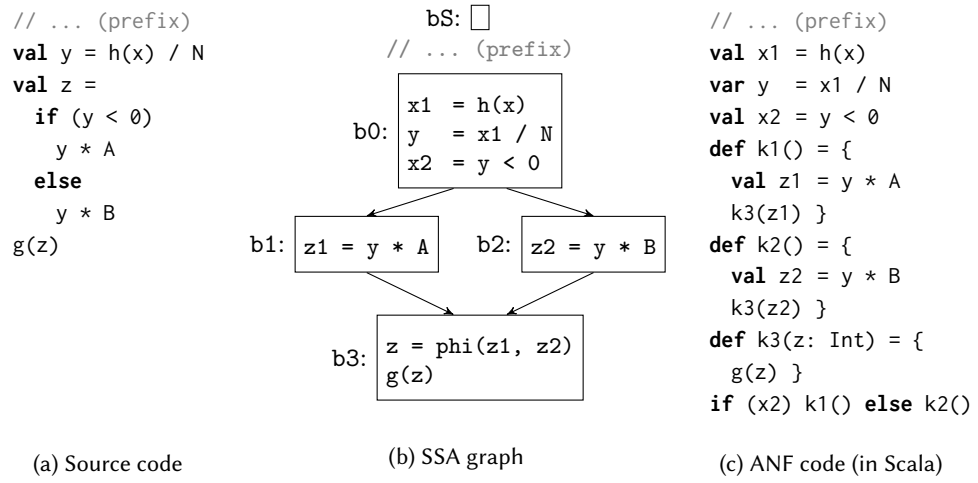
Formally, a comprehension [$e \mid qs$] consists of a *head* expression e and a *qualifier* sequence qs. A qualifier can be either a *generator* $x \leftarrow xs$ binding each element of $xs : \texttt{Bag}[A]$ to $x : A$, or a boolean *guard* p. Monad comprehension semantics can be defined in terms of the *monad with zero* interface. Here, we use a variant of the $\mathcal{MC}$ translation scheme proposed by Grust [36].

$$\begin{aligned} \mathcal{MC}\ [\ e \mid\ ] &= sng(\mathcal{MC}\ e) \\ \mathcal{MC}\ [\ e \mid q, qs\ ] &= flatten(\mathcal{MC}\ [\ \mathcal{MC}\ [\ e \mid qs\ ] \mid q\ ]) \\ \mathcal{MC}\ [\ e \mid x \leftarrow xs\ ] &= map(\lambda x.\ \mathcal{MC}\ e)(\mathcal{MC}\ xs) \\ \mathcal{MC}\ [\ e \mid p\ ] &= \texttt{if}\ \mathcal{MC}\ p\ \texttt{then}\ sng\ (\mathcal{MC}\ e)\ \texttt{else}\ emp \\ \mathcal{MC}\ e &= e \end{aligned} \tag{$\mathcal{MC}$}$$

As a programming language construct, comprehensions were first adopted by Haskell. Nowadays, comprehension syntax is also natively supported by programming languages such as Python (as *list comprehensions*) or Scala (as `for`-*comprehensions*). In our work, we use Scala's ability to support `for`-comprehensions for any user-defined class that implements a *monad with zero* interface consisting of the functions `map`, `flatMap` and `withFilter`.

---

to a homomorphism between the *free commutative monoid* over A – (Bag[A], *emp*, *uni*), and the commutative monoid (*B*, *zero*, *plus*). This view is adopted by Fegaras [21–24], who uses homomorphisms over monoids / commutative monoids / commutative idempotent monoids instead of folds over lists / bags/ sets.



(a) Source code (b) SSA graph (c) ANF code (in Scala)

Fig. 2. Example program.

4.2 Static Single Assignment (SSA) Form

Language compilers typically perform optimizations conditioned on analysis information derived from the data- and control-flow structure of the underlying program. An IR facilitating this kind of analysis therefore is a necessary prerequisite for any optimizing compiler. Since the beginning of the 1990s, SSA and its functional encoding – Administrative Normal Form (ANF) – have been successfully used in a number of compilers. As the IR proposed in Section 6 builds on ANF, this section introduces the main ideas behind SSA and ANF based on a simple example (Figure 2). For a more thorough primer of these concepts, we refer the reader to the overview paper by Appel [5].

The source code formulation of the example program (Figure 2a) offers various degrees of syntactic freedom. For instance, we could have inlined `y` in its call sites, or defined `z` as a variable assigned in the two branches. Program analysis on top of the source-code AST needs to accommodate for these degrees of freedom. In contrast, the derived SSA graph (Figure 2b) offers a normalized representation where data- and control-flow information is encoded directly.

The defining properties of the SSA form are that (*i*) every value is assigned only once, and (*ii*) every assignment abstracts over exactly one function application. In the SSA version of our example, the subexpression `h(x)` is assigned to a fresh variable `x1` and referenced in the division application bound to `y`. Control-flow dependent values are encoded as `phi` nodes. In our example, `z = phi(z1, z2)` indicates that the value of `z` corresponds to either `z1` or `z2`, depending on the input edge along which we have arrived at the `b3` block at runtime.

The SSA graph can be also represented as a functional program in ANF (Figure 2c). In this representation, control-flow blocks are encoded as nested Scala functions referred to as *continuations* (e.g., `k1` through `k3`), and control-flow edges are encoded as calls of these functions (e.g, `k3(z1)` or `k3(z2)`). Values bound to the same continuation parameter correspond to `phi` nodes. For example, `z1` and `z2` are bound to the `z` parameter of `k3` in Figure 2c, corresponding to the `z = phi(z1, z2)` definition in Figure 2b. In addition, the dominance tree associated with an SSA graph is reflected in the nesting structure of the corresponding ANF representation. For example, the `k1`, `k2`, and `k3` continuation definitions are all nested in `k0` (the continuation enclosing the ANF code in Figure 2c).

This implies that the b0 block dominates blocks b1, b2, and b3 – that is, every path rooted at the origin of the SSA graph bS that ends at b1, b2, or b3, must also go through b0.

5 SOURCE LANGUAGE AND PROGRAMMING ABSTRACTIONS

To address the problems outlined in Section 2 we propose *Emma* – a quote-delimited DSL embedded in Scala. Section 5.1 discusses syntactic forms and restrictions driving our design. Based on those, in Section 5.2 we derive a formal definition of *Emma Source* – a subset of Scala accepted by the *Emma* compiler. Finally, Section 5.3 presents the programming abstractions forming the *Emma* API.

5.1 Syntactic Forms and Restrictions

In Section 3, we claimed that problems with state-of-the-art EDSLs for parallel collection processing are a consequence of the adopted type-based embedding strategy, as the program structure critical for optimization is either not represented or is treated as a black box in the DSL IR.

We analyzed a wide range of algorithms implemented in the RDD and DataSet EDSLs and identified the following base set of relevant syntactic forms: **(F1)** if-else, while, and do-while control-flow primitives, **(F2)** var and val definitions and var assignments, **(F3)** lambda function definitions, **(F4)** def method calls and new object instantiations, and **(F5)** statement blocks. The ability to freely compose those forms in the host language (Scala) and reflect them in the IR is crucial in order to attain maximal syntactic reuse without limiting the EDSL optimization potential.

In addition to (**F1**-**F5**), the following forms are either defined in the Scala ASTs in terms of (**F1**-**F5**), or can be eliminated with a simple ASTs transformation. **(F6)** for-comprehensions – those are represented as chains of nested flatMap, withFilter, and map calls based a DESUGAR scheme similar to the $\mathcal{MC}$ transformation from Section 4.1 which is implemented by the Scala compiler. **(F7)** irrefutable patterns (that is, patterns that are statically guaranteed to always match) – those can be transformed in terms of val definitions and def calls. **(F8)** for loops – those are rewritten as foreach calls by the Scala compiler and can be subsequently transformed into while loops.

Finally, we made some restrictions in order to simplify the compiler frontend and the optimizing program transformations presented in the rest of this paper. The following syntactic forms therefore are not supported by *Emma*: **(R1)** def definitions, **(R2)** lazy and implicit val definitions, **(R3)** refutable patterns, **(R4)** call-by-name parameters, **(R5)** try-catch blocks, **(R6)** calls of referentially opaque (that is, effectful) def methods, and **(R7)** var assignments outside of their defining scope (i.e. inside a lambda). All restrictions except **R6** can be asserted by an additional pass of the Scala AST representing the quoted code fragment. Since Scala does not provide a built-in effect system, adherence to **R6** is assumed as given and is based on developer discipline.

5.2 Source Language Syntax

We proceed by formalizing *Emma Source* – a user-facing language which models a subset of Scala covering (**F1**-**F5**) as abstract syntax that can be derived from quoted Scala ASTs.

The specification presented below relies on the following terminology and notational conventions. *Metaprogramming* is the ability of computer programs to treat other programs as data. The language in which the *metaprogram* is written is called *metalanguage*, and the language being manipulated – *object language*. *Reflection* is the ability of a programming language to act as its own metalanguage. *Emma Source* models a subset of Scala, and (since it is an embedded DSL) the metalanguage is also Scala. We use Scala's compile- and runtime reflection capabilities to implement the compiler infrastructure presented in the next sections.

$t :=$		term	$stat :=$		statement
	$t.m[T_j](ts)_i$	method call		$x = t$	assignment
	`new` $X[T_j](ts)_i$	`new` instance		*loop*	loop
	$pdefs \Rightarrow t$	lambda		*bdef*	binding def
	$t.module$	mod. access	$loop :=$		loop
	$t : T$	ascription		`while` (t) *block*	`while`
	`if` (t_1) t_2 `else` t_3	conditional		`do` *block* `while` (t)	`do-while`
	{ *stats*; t }	stats block	$bdef :=$		binding def
	a	atomic		`val` $x : T = t$	`val` def
$a :=$		atomic		`var` $x : T = t$	`var` def
	lit	literal		*pdef*	param. def
	this	`this` ref	$pdef :=$	$x : T$	param. def
	module	module ref			
	x	binding ref			

Fig. 3. Abstract syntax of *Emma Source*.

We denote metalanguage expressions in *italic* and object-language expressions in a `teletype` font family. Syntactic forms in the object language may be parameterized over metalanguage variables standing for other syntactic forms. For example, t`.take(10)` represents an object-language expression where t ranges over object-language terms like `xs` or `ys.tail`. A metalanguage name suffixed with s denotes a sequence, and an indexed subexpression a repetition. For example $(ts)_i$ denotes repeated term sequences enclosed in parentheses.

The abstract syntax of *Emma Source* is listed in Figure 3. It consists of two mutually recursive definitions – *terms* (which always return a value), and *statements* (which modify the computation state). In the following paragraphs, we discuss some important aspects of *Emma Source*.

Note that *Emma Source* offers mechanisms for both (i) abstraction (lambda terms) and (ii) control-flow (conditional terms and loop constructs). Crucially, the proposed abstract syntax ensures that (ii) is stratified with respect to (i). This assumption would be violated if recursive functions (`def` definitions in Scala) were included in *Source*. This restriction simplifies the decision procedure for the concept of *binding context* (see Section 6.5).

5.3 Programming Abstractions

The core programming abstraction is a trait `Bag[A]` representing a distributed collection with elements of type `A`, and a matching `BagCompanion` trait defining `Bag` constructors (Figure 4). To illustrate the differences between the `Bag` and the `RDD`/`DataSet` APIs, we re-cast examples from Section 2.3. Note that all syntactic issues outlined in Section 2.3 are resolved in the `Bag` API.

5.3.1 Sources and Sinks. The *data sources* in the `BagCompanion` trait define various `Bag` constructors. For each *source* there is a corresponding *sink* operating in the reverse direction.

```
val movies  = Bag.readCSV[Person]("hdfs://.../movies.csv", ...) // from file
val credits = Bag.from(creditsRDD) // from a Spark RDD / Flink DataSet
val people  = Bag.apply(peopleSeq) // from a local Scala Seq
```

Data sinks (in Bag[A])

```
fetch() : Seq[A]
as[DColl[_]] : DColl[A]
writeParquet(path : String, ...) : Unit
writeCSV(path : String, ...) : Unit
```

Data sources (in BagCompanion)

```
empty[A] : Bag[A]
apply[A](values : Seq[A]) : Bag[A]
from[DColl[_], A](coll : DColl[A]) : Bag[A]
readParquet[A](path : String, ...) : Bag[A]
readCSV[A](path : String, ...) : Bag[A]
```

SQL-like (in Bag[A])

```
map[B](f : A ⇒ B) : Bag[B]
flatMap[B](f : A ⇒ Bag[B] : Bag[B]
withFilter(p : A ⇒ Boolean) : Bag[A]
union(that : Bag[A]) : Bag[A]
groupBy[K](k : A ⇒ K) : Bag[Group[K, Bag[A]]]
distinct : Bag[A]
```

Folds (in Bag[A])

```
fold[B](alg : Alg[A, B]) : B
size, nonEmpty, min, max, ...
```

Fig. 4. Bag[A] and BagCompanion API.

5.3.2 Select-From-Where-like Syntax. The operators in the right column in Figure 4 enable a Scala-native API for collection processing similar to SQL. Note that binary operators like `join` and `cross` are omitted from the API. Instead, `Bag` implements the monad interface discussed in Section 4.1. This allows for `Select-From-Where`-like expressions using Scala's `for`-comprehension syntax. The join chain from Example 2.2 can be expressed as follows.

```
val ys = for {
  m <- movies; c <- credits; p <- people
  if m.id == c.movieID; if p.id == c.personID
} yield (m.title, p.name)
```

We keep the above syntax at the IR level and employ rule-based query compilation heuristics such as filter-pushdown and join-order optimization (see Section 7).

5.3.3 Aggregation and Grouping. `Bag` aggregations are based on structural recursion over Union-style bags. The `fold` method accepts a Union-algebra that encapsulates the substitution functions for the three bag constructors. The algebra trait `Alg` and an example instance algebra `Size` that counts the number of elements in the input collection are defined as follows.

```
trait Alg[-A, B] {
  val zero: B
  val init: A => B
  val plus: (B, B) => B
}
```

```
object Size extends Alg[Any, Long] {
  val zero = 0L
  val init = (x: Any) => 1L
  val plus = (x: Long, y: Long) => x + y
}
```

Note that the `Alg` type definition is contravariant in the element type `A` – if `X` is a subtype of `Y`, for a fixed result type `B` the type `Alg[Y, B]` will be a subtype of `Alg[X, B]`. Contravariance allows us to define algebras of type `Alg[Any, B]` (such as `Size`), which can be used to fold bags with arbitrary element type (in Scala, all types are a subtypes of `Any`). Common folds are aliased as dedicated methods. For example, `xs.size` is defined as follows.

```
def size: Long = this.fold(Size) // using the 'Size' algebra from above
```

The groupBy method returns a Bag of Group instances, where the Group class consists of a group key of type K and group values of type Bag[A] (where A is the element type of the input bag). Per-group aggregates are defined in terms of a groupBy and a for-comprehension. In the following example, we also use pattern matching in the left-hand side of the comprehension generator in order to directly extract the key (d) and values (ms) of each group.

```
for {
  Group(d, ms) <- movies.groupBy(decade(_.year))
} yield (d, ms.size)
```

Rewriting this definition in terms of operators such as reduceByKey is enabled by (*i*) the insight that folds over Union-style collections model data-parallel computation, and (*ii*) the ability to represent nested Bag computations in the IR (see Section 8).

5.3.4 Caching and Native Iterations. The Bag API does not require explicit caching. Bag terms referenced inside a loop or more than once are cached implicitly (Section 9). For example, in

```
val S = static()
var w = init() // outer `w`
for (i <- 0 until N) {
  w = update(S, w) // inner `w`
}
```

S and the inner w are cached. In addition, we propose a transformation that rewrites loop structures to Flink's iterate operator whenever possible (Section 10).

5.3.5 API Implementations. The Bag and BagCompanion traits are implemented once per backend. Current implementations are ScalaBag (backed by a Scala Seq), FlinkBag (backed by a Flink DataSet) and SparkBag (backed by either a Spark Dataset or a Spark RDD). The ScalaBag implementation is used per default (constructors in Bag companion just delegate to the ScalaBag companion object), while one of the other two implementations is introduced transparently as part of the compilation pipeline as sketched in Section 6.6.

Unquoted *Emma* code therefore can be executed and debugged as regular Scala programs. Consequently, developers can focus on writing semantically correct code first, and quote the *Emma Source* code snippet in order to parallelize it later.[7]

6 CORE LANGUAGE AND NORMALIZATION

As a basis for the optimizations presented in the next sections we propose an IR called *Emma Core*. Section 6.1 presents an ANF subset of this IR called *Emma Core*$_{ANF}$ together with a translation scheme from *Emma Source* to *Core*$_{ANF}$. To accommodate for SQL-like program rewrites, Section 6.2 incorporates first-class monad comprehensions, extending *Emma Core*$_{ANF}$ to *Emma Core*, and Section 6.3 sketches a comprehension normalization scheme. Section 6.4 illustrates how *Emma Core* can be used to check conditions required for the optimizations presented in Sections 7 through 10. Section 6.5 describes a transformation that specializes the execution backend of top-level Bag expressions, and the related notion of *binding context*. Finally, Section 6.6 gives an overview of the *Emma* compiler pipeline, putting all the pieces presented in Section 6 together.

[7]The net effect is similar to the difference between deep and shallow embeddings of EDSLs implemented on top of the Yin-Yang DSL framework [42].

$a :=$	... (as in Figure 3)	atomic	$let :=$	{ $vdefs$; $kdefs$; c }	let block
$b :=$		binding			
	$a.m[T_j](as)_i$	method call	$kdef :=$	`def` $k(pdefs) = let$	cont. def
	`new` $X[T_j](as)_i$	`new` instance	$bdef :=$		binding def
	$pdefs \Rightarrow let$	lambda		$vdef$	`val` def
	$a.module$	mod. access		$pdef$	param. def
	$a : T$	ascription	$vdef :=$	`val` $x = b$	val def
	a	atomic	$pdef :=$	$x : T$	param. def
$c :=$		cont. call			
	`if` (a) $k(as)$ `else` $k(as)$	branching			
	$k(as)$	simple			
	a	atomic			

Fig. 5. Abstract syntax of *Emma Core$_{ANF}$*.

6.1 Core ANF Language

The abstract syntax of the *Emma Core$_{ANF}$* language is specified in Figure 5. Below, we outline the main differences between *Emma Core$_{ANF}$* and *Emma Source*.

The sub-language of atomic terms (denoted by a) is shared between the two languages, while imperative statement blocks are replaced by functional *let* blocks. To ensure that all sub-terms (except lambda) are atomic, terms that may appear on the right-hand side of `val` definitions are restricted from t to b. Control-flow statements are replaced by continuation functions in the so-called *direct-style*, and `var` definitions and assignments by continuation parameters. Continuations may only appear after the `val` sequence in *let* blocks and can be called only in the c position.

The DSCF $\circ$ ANF : *Source* $\Rightarrow$ *Core$_{ANF}$* translation is defined in terms of two composed transformations – ANF : *Source* $\Rightarrow$ *Source$_{ANF}$* and DSCF : *Source$_{ANF}$* $\Rightarrow$ *Core$_{ANF}$*. The complete set of inference rules for these transformations can be found in the electronic appendix.

The ANF transformation destructs compound t terms as statement blocks where each sub-term becomes a named b-term bound to a `val` definition and the return expression is an a-term (that is, atomic). Terms appearing on the right-hand-side of `var` definitions and assignments are always atomic. The resulting language is denoted as *Source$_{ANF}$*. To illustrate, consider the expression

$$\text{ANF}[\![\ \{\ \texttt{z = x * x + y * y;\ Math.sqrt(z)}\ \}\ ]\!]$$

which results in a statement block that encodes the def-use dependencies of the original program.

$$\{\ \texttt{val u}_1 = \texttt{x * x;\ val u}_2 = \texttt{y * y;\ val u}_3 = \texttt{u}_1 + \texttt{u}_2\texttt{;\ z = u}_3\texttt{;\ val u}_4 = \texttt{Math.sqrt(z)\ ;\ u}_4\ \}$$

The translation from *Source$_{ANF}$* to *Core$_{ANF}$* is handled by the DSCF transformation. For terms $t' = \text{ANF}[\![t]\!]$ that do not contain `var` definitions, assignments, and control-flow statements, DSCF$[\![t']\!]$ will simply convert all *stats* blocks in ANF$[\![t']\!]$ to *Core$_{ANF}$ let* blocks. To eliminate variables, the rules DSCF-VAR and DSCF-ASGN (which is structurally similar to DSCF-VAR) accumulate an environment $\mathcal{V}$ that keeps track of the most recent atomic term a associated with each variable x and maps

those in rule DSCF-REF2.

$$\textsc{dscf-var}\ \frac{\mathcal{V} \vdash a \mapsto a' \qquad \mathcal{V}, x \leftarrow a' \vdash \{\ ss;\ c\ \} \mapsto let}{\mathcal{V} \vdash \{\ \texttt{var}\ x = a; ss;\ c\ \} \mapsto let} \qquad \frac{\mathcal{V} x = a}{\mathcal{V} \vdash x \mapsto a}\ \textsc{dscf-ref2}$$

Loops and conditionals are handled by DSCF-IF1, DSCF-IF2, DSCF-WDO, and DSCF-DOW.

$$\textsc{dscf-dow}\ \frac{x_i \in \mathcal{A}[\![\texttt{do}\ \{\ ss_2;\ a_2\ \}\ \texttt{while}\ (\{\ ss_1;\ a_1\ \})]\!] \qquad \mathcal{V} x_i = a_i' \qquad \mathcal{V}, x_i \leftarrow p_i \vdash \{\ ss_3;\ c_3\ \} \mapsto let_3 \qquad \mathcal{V}, x_i \leftarrow p_i \vdash \{\ ss_1; ss_2; \texttt{def}\ k_3() = let_3;\ \texttt{if}\ (a_2)\ k_1(x_i)\ \texttt{else}\ k_3()\ \} \mapsto let_1}{\mathcal{V} \vdash \{\ \texttt{do}\ \{\ ss_2;\ a_2\ \}\ \texttt{while}\ (\{\ ss_1;\ a_1\ \}); ss_3;\ c_3\ \} \mapsto \{\ \texttt{def}\ k_1(p_i) = let_1;\ k_1(a_i')\ \}}$$

The antecedents of these rules rely on two auxiliary functions: $\mathcal{R}[\![t]\!]$ computes the set of binding symbols referenced in t, while $\mathcal{A}[\![t]\!]$ computes the set of variable symbols assigned in t. A variable x_i that is assigned in a matched control-flow form is converted to a parameter p_i in the corresponding continuation definition. Handling of conditionals with a general form

$$\{\ \texttt{val}\ x = \texttt{if}\ (a)\ \{\ ss_1;\ a_1\ \}\ \texttt{else}\ \{\ ss_2;\ a_2\ \}; ss_3;\ c_3\ \}$$

diverges based on $x \in \mathcal{R}[\![\{\ ss_3;\ c_3\ \}]\!]$ – if x is referenced in the suffix, the signature and the calls of the corresponding k_3 continuation need to be adapted accordingly.

The DSCF rewrite also asserts the certain properties of the resulting trees. First, the parent-child relationship of the nested continuation function definitions encodes the dominator tree of the control-flow graph. Second, continuation functions do not contain parameters that always bind to the same argument. Third, with exception of the terms in nested lambda bodies, the resulting term t has exactly one *let* block of the form { vals ; a }, denoted $\textsc{suffix}[\![t]\!]$.

6.2 Adding First-Class Monad Comprehensions

An IR for *Emma* should facilitate common optimizations from the domains of language and query compilation. While *Emma Core*$_{ANF}$ is a good fit for the first, query compilation starts with a `Select-From-Where`-like expression and ends with a relational algebra expression. As a basis for this transformation, a query compiler typically uses a join graph derived from the `Select-From-Where` expression [27, 50, 55]. Respecting the correspondence between `Select-From-Where` expressions and `for`-comprehensions, our goal is to enable similar techniques in the *Emma* IR. However, `for`-comprehensions in the Scala AST and the derived *Core*$_{ANF}$ are encoded as chains of nested `flatMap`, `withFilter`, and `map` applications. To bridge the gap, we add support for first-class monad comprehensions to *Core*$_{ANF}$.

The resulting language, called *Emma Core*, is depicted on Figure 6. Similar to *Emma*$_{ANF}$ lambda bodies, sub-terms in comprehension heads, generator right-hand-sides, and guard expressions are restricted to be *let* blocks. This simplifies the development of *Core*-based optimizations without loss of generality, as a terms can be canonically encoded as { a } and b terms as { `val` x = b ; x }.

The translation from *Emma Core*$_{ANF}$ to *Emma Core* proceeds in two steps. First, we apply the RESUGAR$_{\texttt{Bag}}$ transformation, converting `flatMap`, `withFilter`, and `map` calls on `Bag` targets to

$b :=$	...	binding term
	`for` { qs } `yield` *let*	comprehension

$q :=$		qualifier
	$x \leftarrow$ *let*	generator
	`if` *let*	guard

Fig. 6. Extending the abstract syntax of *Emma Core*$_{ANF}$ to *Emma Core*.

$$\text{RES-MAP}\ \frac{\mathcal{X} f = x : A \Rightarrow \mathit{let} \qquad a : \mathsf{M}A}{\mathcal{X} \vdash a.\mathtt{map}(f) \mapsto \mathtt{for}\ \{\ x \leftarrow a\ \}\ \mathtt{yield}\ \mathit{let}}$$

$$\text{RES-FMAP}\ \frac{\mathcal{X} f = x_1 : A \Rightarrow \mathit{let} \qquad a : \mathsf{M}A}{\mathcal{X} \vdash a.\mathtt{flatMap}(f) \mapsto \mathtt{for}\ \{\ x_1 \leftarrow a; x_2 \leftarrow \mathit{let}\ \}\ \mathtt{yield}\ \{\ x_2\ \}}$$

$$\text{RES-FILTER}\ \frac{\mathcal{X} f = x : A \Rightarrow \mathit{let} \qquad a : \mathsf{M}A}{\mathcal{X} \vdash a.\mathtt{withFilter}(f) \mapsto \mathtt{for}\ \{\ x \leftarrow a; \mathtt{if}\ \mathit{let}\ \}\ \mathtt{yield}\ \{\ x\ \}}$$

Fig. 7. Main inference rules for the $\text{RESUGAR}_{\mathsf{M}} : \mathit{Core}_{\mathit{ANF}} \Rightarrow \mathit{Core}$ transformation. The type former M should be a monad, i.e., it should implement `map`, `flatMap`, and `withFilter` obeying the "monad with zero" laws.

simple monad comprehensions (Figure 7). Resulting simple comprehensions that are nested in each other are then combined into bigger comprehensions by the $\text{NORMALIZE}_{\mathsf{Bag}}$ transformation.

In $\text{RESUGAR}_{\mathsf{Bag}}$, rule application depends on a context $\mathcal{X}$ of available lambda definitions, accumulated and operating in line with the following definition.

$$\mathcal{X} f = \begin{cases} x : A \Rightarrow \mathit{let} & \text{if } \mathtt{val}\ f = x : A \Rightarrow \mathit{let} \text{ is in the current scope} \\ x : A \Rightarrow \{\ \mathtt{val}\ x' = f(x);\ x'\ \} & \text{otherwise} \end{cases}$$

If f is not a *lambda* defined in the current scope, $\mathcal{X}$ will associate f with an eta-expansion of itself – that is, with a *lambda* that just applies f to its parameter. This allows to not only resugar terms representing desugared `for`-comprehensions, but also cover terms like `xs.withFilter(isOdd)`, even if `isOdd` is not defined in the quoted code fragment.

6.3 Comprehension Normalization

We proceed with a normalization that repeatedly merges def-use chains of smaller comprehensions. The normalizing transformation $\text{NORMALIZE}_{\mathsf{M}}$ repeatedly applies the UNNEST-HEAD rule (Figure 8) until convergence. The consequent matches an enclosing *let* block that contains an MA comprehension definition identified by x_1, with a generator symbol x_3 that binds values from x_2. The rule triggers if x_2 binds to a comprehension in vdefs_1 or vdefs_2 and is not referenced otherwise. The rewrite depends on the auxiliary functions *split*, *fix* and *remove*. A $\mathit{remove}(x, \mathit{vdefs})$ application removes a value definition `val` $x = b$ from *vdefs*. An application of $\mathit{split}(\mathit{vdefs}, \mathit{qs})$ partitions *vdefs* into two subsequences – vdefs^D and vdefs^I, which respectively (transitively) depend and do not depend on generator symbols defined in *qs*. Finally, $\mathit{fix}(e)$ where $e = x \leftarrow \mathit{let} \mid \mathtt{if}\ \mathit{let} \mid \mathit{let}$ adapts $\mathit{let} = \{\ \mathit{vals};\ \mathit{defs};\ c\ \}$ in two steps. First, it obtains let' by inlining let_2 (which contains the x_3 definition) in *let*. If $x_3 \notin \mathcal{R}[\![\mathit{let}]\!]$, we have $\mathit{let}' = \mathit{let}$, otherwise let' is derived from let_2 by extending $\text{SUFFIX}[\![\mathit{let}_2]\!] = \{\ \mathit{vals}_S;\ a_S\ \}$ as $[x_3 := a_S]\{\ \mathit{vals}_S; \mathit{vals};\ \mathit{defs};\ c\ \}$. Second, copies of the dependent values vdefs_2^D referenced in *let* are prepended to let'.

6.4 Utility of the Core Language

The *Core* language defined below is an extension of the ANF representation presented in Section 4.2. This simplifies program analysis and ensures that we can check the necessary conditions for the program transformations developed on top of *Emma Core* in an efficient and robust manner.

$$\frac{\begin{array}{c}\textsc{UnnestHead}\\ x_1 : \mathsf{M}A \qquad \texttt{val}\ x_2 = \texttt{for}\ \{\ qs_2\ \}\ \texttt{yield}\ let_2 \in vdefs_1 +\!\!+ vdefs_2 \qquad uses(x_2) = 1\\ (vdefs_2^I, vdefs_2^D) := split(remove(x_2, vdefs_2), qs_1)\\ qs' := qs_1 +\!\!+ qs_2 +\!\!+ qs_3.map(fix) \qquad let_1' := fix(let_1) \qquad vdefs_1' := remove(x_2, vdefs_1)\end{array}}{\begin{array}{c}\{\ vdefs_1; \texttt{val}\ x_1 = \texttt{for}\ \{\ qs_1; x_3 \leftarrow \{\ vdefs_2;\ x_2\ \}; qs_3\ \}\ \texttt{yield}\ let_1; vdefs_3;\ kdefs;\ c\ \}\\ \mapsto \{\ vdefs_1'; vdefs_2^I; \texttt{val}\ x_1 = \texttt{for}\ \{\ qs'\ \}\ \texttt{yield}\ let_1'; vdefs_3;\ kdefs;\ c\ \}\end{array}}$$

Fig. 8. The UNNEST-HEAD rule used in the $\textsc{normalize}_{\mathsf{M}} : Core \Rightarrow Core$ transformation. As in Figure 7, M can be any type former which is also a monad.

For example, the FOLD-GROUP-FUSION transformation discussed in Section 8.2 can only be applied if the values resulting from a `groupBy` application are used exactly once, and this use is within the context of a `fold` application. Based on the ANF characteristics of *Emma Core*, this condition can be checked as follows. Step (1): look for a *vdef* matching the pattern $\texttt{val}\ x_1 = a_1.\texttt{groupBy}(a_2)$. Step (2): look for a generator pattern $x_2 \leftarrow x_1$ which binds individual `Group` instances from x_1 to x_2. Step (3): find the symbol x_3 associated with the group values using the pattern $\texttt{val}\ x_3 = x_2.\texttt{values}$. Step (4): if x_3 exists, find all *vdefs* $\texttt{val}\ x_4 = b$ where x_3 occurs in the binding b. Step (5): The condition is satisfied if there is only one such b and it has the form $x_3.\texttt{fold}(a_3)$. The atomic term a_3 represents the algebra instance that needs to be fused with the `groupBy` call from Step (1).

6.5 Backend Specialization

The `Bag` API allows for nesting. Nested bags can be constructed either as a result of a `groupBy` application or directly. For example, if the head of the following comprehension

$$\texttt{val ts = for \{ d} \leftarrow \texttt{docs \} yield \{}\ vdefs;\ \texttt{tokens \}}$$

constructs a value `tokens` of type `Bag[String]`, the type of `ts` will be `Bag[Bag[String]]`. After translating this `for`-comprehension to an equivalent dataflow graph (see Section 7), we will have the following value definitions: $\texttt{val ts = docs.map(f)}$ and $\texttt{val f = (d : String)} \Rightarrow \{\ vdefs;\ \texttt{tokens}\ \}$.

Bag nesting leads to better programming experience, but poses some compile-time challenges. Recall that the `Bag` and `BagCompanion` APIs are implemented once per backend – a `SparkBag` and a `FlinkBag`. A naïve way to automatically offload `Bag` computations to a dataflow engine is to substitute all `Bag` companion constructor calls with the appropriate backend companion. For the above example and a Spark backend, this means using the `SparkBag` companion for the `docs` and `tokens` constructor calls. The problem with this approach is that the `tokens` constructor call is part of the `f` lambda *vdefs*, and `f` will be executed in the context of a Spark worker due to the `docs.map(f)` call. This will cause a runtime error – the resulting `SparkBag` instance is backed by a Spark `Dataset`, but the Spark runtime prohibits the construction of `Dataset` and `RDD` instances on worker nodes. To overcome this problem, we have to specialize only those `Bag` constructor calls that are not nested within `Bag` expressions. To be able to do that, we need to distinguish between `Bag` symbols whose value is bound in a top-level context (i.e., in the context of the *driver* program orchestrating the execution of the parallel dataflows) from the ones bound in the context of a dataflow *engine* (either in Spark Worker Node or in a Flink TaskManager).

Definition 6.1 (Binding Context). The *binding context* of a binding symbol x, denoted $C(x)$, is a value from the $\{Driver, Engine, Ambiguous\}$ domain that identifies the context in which that symbol might be bound to a value at runtime.

```
val f = (doc: Document) => {
  // ... extract `brands` from `doc`
  brands
}
val bs = f(d0)
val rs = for {
  d <- docs
  b <- f(d)
  if bs contains b
} yield d
```

(a) *Emma Source* snippet

$$C(x) = \begin{cases} \textit{Driver} & \text{if } x \in \{\, \texttt{f}, \texttt{bs}, \texttt{rs} \,\} \\ \textit{Engine} & \text{if } x \in \{\, \texttt{d}, \texttt{b} \,\} \\ \textit{Ambiguous} & \text{if } x \in \{\, \texttt{doc}, \texttt{brands} \,\} \end{cases}$$

(b) computed binding context values

Fig. 9. Binding context example.

To compute $C(x)$ for all symbols x defined in a *Emma Core* term t we use a procedure called CONTEXT$[\![t]\!]$. To explain how CONTEXT works, consider the example from Figure 9a. We first define a function `f` that extracts `brands` mentioned in a document `doc`. We then use `f` to compute the `Bag` of brands `bs` mentioned in a seed document `d0`. Finally, from the `Bag` of documents `docs` we select documents `d` mentioning a brand `b` contained in `bs`. The result of the CONTEXT procedure for this example snippet is depicted on Figure 9b. The C-value of symbols defined in the outer-most scope (e.g. `f`, `bs`, and `rs`) is always *Driver*. The C-value of generator symbols (such as `d` and `b`) is always *Engine*. The C-value of symbols nested in lambdas, however, depends the lambda uses. For example, `f` is used both in a *Driver* context (the `bs` definition) and in an *Engine* context (the `rs` definition). Consequently, the C-value of all symbols defined in the `f` lambda (such as `doc` and `brands`) is *Ambiguous*. The context of nested lambdas is computed recursively.

We want to specialize the definitions of terms that denote `Bag` constructor applications and are evaluated in the driver. Conservatively, we prohibit programs where such terms have *Ambiguous* binding context. Compilation in our running example will fail because $C(\texttt{brands}) = \textit{Ambiguous}$. To alleviate this restriction, one can duplicate lambdas with ambiguous use (such as `f`) and disambiguate their call sites. Here, we opted for a more restrictive, but simpler approach because the programs implemented so far did not contain symbols with *Ambiguous* context.

As part of the backend specialization routine, we also create broadcast versions xs' of all *Driver* bags xs which are used in an *Engine* context and substitute xs with xs' in the *Engine* use sites.

6.6 Compiler Pipelines

The transformations presented in this chapter form the basis of the following compiler frontend.

$$\text{LIFT} \quad = \quad \text{NORMALIZE}_{\texttt{Bag}} \circ \text{RESUGAR}_{\texttt{Bag}} \circ \text{DSCF} \circ \text{ANF}$$

Quoted *Emma Source* terms are first lifted to *Emma Core* by this frontend pipeline. The resulting term is then iteratively transformed by a chain of *Core* $\Rightarrow$ *Core* optimizing transformations such as the ones discussed in Sections 7 through 10.

$$\text{OPTIMIZE} \quad = \quad \text{OPTIMIZE}_n \circ \ldots \circ \text{OPTIMIZE}_1$$

While individual optimizing transformations might be defined in a backend-agnostic way, the concrete OPTIMIZE chain is backend-dependent, as it contains at least one backend-specific transformation (e.g., native iteration specialization for Flink or structured API specialization in Spark).

For both backends, the chain ends with a transformation that specializes the backend. This transformation identifies *vdef* terms of the form `val` x = `Bag.`m`[...](...)` , where $C(x)$ = *Driver* and m matches one of the source methods listed in Figure 4. The `Bag` companion object in the matched *vdef* is then specialized either as `SparkBag` or `FlinkBag`.

In contrast to other functional programming languages, Scala eliminates tail calls only in self-recursive methods. To avoid stack overflow errors at runtime, each compiler pipeline ends with an inverse DSCF transformation. The transformation converts continuation definitions to control-flow statements (such as `while` loops), and continuation parameters to *var* definitions and assignments.

We end up with two different basic pipelines defined as Scala macros – one for Spark (named `onSpark`) and one for Flink (named `onFlink`). Quoting (that is, enclosing) a Scala code fragment in one of these macros executes the corresponding pipeline at compile time.

We also add a `lib` macro-annotation which can be used to annotate objects that define library functions. Quoted calls to these functions are inlined recursively before the LIFT transformation, and lambdas used only once are β-reduced before the OPTIMIZE step. *Emma* programmers therefore can structure their code as modular and composable libraries. At the same time, optimizing quoted data analysis pipelines after inlining of `lib` methods and β-reduction ensures that the optimization potential of the *Emma* compiler is always fully preserved.

7 COMPREHENSION COMPILATION

The core language presented in Section 6 integrates `Bag` comprehensions as a first-class syntactic form. The *Emma* compiler has to transform the normalized `Bag` comprehensions into dataflow expressions based on the operators supported by the targeted parallel dataflow API.

7.1 Naïve Strategy

A naïve strategy following the $\text{DESUGAR}_{\text{Bag}}$ scheme (see **F6** in Section 5.1) can lead to suboptimal dataflows. To see why, let e denote a comprehension defining an equi-join between two `Bag` terms.

$$\texttt{for } \{\ \texttt{x} \leftarrow xs\ ;\ \texttt{y} \leftarrow ys\ ;\ \texttt{if } \texttt{k}_\texttt{x}(\texttt{x}) = \texttt{k}_\texttt{y}(\texttt{y})\ \}\ \texttt{yield } (\texttt{x}, \texttt{y})$$

Then $\text{DESUGAR}_{\text{Bag}}[\![e]\!]$ denotes the following term.

$$xs.\texttt{flatMap}(\texttt{x} \Rightarrow ys.\texttt{withFilter}(\texttt{y} \Rightarrow \texttt{k}_\texttt{x}(\texttt{x}) = \texttt{k}_\texttt{y}(\texttt{y})).\texttt{map}(\texttt{y} \Rightarrow (\texttt{x}, \texttt{y})))$$

The backend specialization transformation will then refine *xs* and *ys* as a `FlinkBag` or a `SparkBag` and substitute the *ys* use within the `flatMap` lambda with a (`ScalaBag`-based) broadcast version of *ys*. The resulting parallel dataflow therefore corresponds to an inefficient broadcast nested-loop join that uses *xs* as a partitioned (outer) and *ys* as a broadcast (inner) relation.

7.2 Qualifier Combination

As demonstrated by the code examples in Section 2.3, however, the targeted parallel dataflow APIs provide dedicated operators for efficient distributed equi-joins. To exploit them, we adopt the approach proposed by Grust [33, 36] and abstract over *equi-join* and *cross* comprehensions with corresponding *comprehension combinator* definitions.

```
def equiJoin[A,B,K](kx: A => K, ky: B => K)(xs: Bag[A], ys: Bag[B]): Bag[(A,B)] =
  for { x <- xs; y <- ys; if kx(x) == ky(y) } yield (x, y)
def cross[A,B](xs: Bag[A], ys: Bag[B]): Bag[(A, B)] =
  for { x <- xs; y <- ys } yield (x, y)
```

Combinator signatures are declared in a `ComprehensionCombinators` trait which is implemented three times. The `LocalOps` implementation contains the above naïve definitions, whereas

$$\textsc{com-filter}\quad \frac{x \in \mathcal{R}[\![p]\!] \qquad \mathcal{R}[\![p]\!] \cap \mathcal{G}[\![qs_1 +\!\!+ qs_2]\!] = \emptyset}{\texttt{for}\ \{\ qs_1;\ x \leftarrow xs;\ qs_2;\ \texttt{if}\ p;\ qs_3\ \}\ \texttt{yield}\ let \mapsto \texttt{for}\ \{\ qs_1;\ x \leftarrow xs.\texttt{withFilter}(x \Rightarrow p);\ qs_2;\ qs_3\ \}\ \texttt{yield}\ let}$$

$$\textsc{com-fmap1}\quad \frac{x \in \mathcal{R}[\![ys]\!] \qquad \mathcal{R}[\![ys]\!] \cap \mathcal{G}[\![qs_2]\!] = \emptyset \qquad x \notin \mathcal{R}[\![qs_2 +\!\!+ qs_3]\!] \qquad x \notin \mathcal{R}[\![let]\!]}{\texttt{for}\ \{\ qs_1;\ x \leftarrow xs;\ qs_2;\ y \leftarrow ys;\ qs_3\ \}\ \texttt{yield}\ let \mapsto \texttt{for}\ \{\ qs_1;\ qs_2;\ y \leftarrow xs.\texttt{flatMap}(x \Rightarrow ys);\ qs_3\ \}\ \texttt{yield}\ let}$$

$$\textsc{com-fmap2}\quad \frac{x \in \mathcal{R}[\![ys]\!] \qquad \mathcal{R}[\![ys]\!] \cap \mathcal{G}[\![qs_2]\!] = \emptyset \qquad t' = [x := z.\_1][y := z.\_2]t}{\texttt{for}\ \{\ qs_1;\ x \leftarrow xs;\ qs_2;\ y \leftarrow ys;\ qs_3\ \}\ \texttt{yield}\ let \mapsto \texttt{for}\ \{\ qs_1;\ z \leftarrow xs.\texttt{flatMap}(x \Rightarrow ys.\texttt{map}(y \Rightarrow (x, y)));\ qs_2';\ qs_3'\ \}\ \texttt{yield}\ let'}$$

$$\textsc{com-join}\quad \frac{\begin{array}{c} x \notin \mathcal{R}^*[\![ys]\!] \qquad x \in \mathcal{R}[\![k_x]\!] \qquad y \in \mathcal{R}[\![k_y]\!] \qquad t' = [x := z.\_1][y := z.\_2]t \\ \mathcal{R}[\![k_y]\!] \cap \mathcal{G}[\![qs_1 +\!\!+ qs_2 +\!\!+ qs_3]\!] = \emptyset \qquad \mathcal{R}[\![k_x]\!] \cap \mathcal{G}[\![qs_1 +\!\!+ qs_2 +\!\!+ qs_3]\!] = \emptyset \end{array}}{\texttt{for}\ \{\ qs_1;\ x \leftarrow xs;\ qs_2;\ y \leftarrow ys;\ qs_3;\ \texttt{if}\ k_x = k_y;\ qs_4\ \}\ \texttt{yield}\ let \mapsto \texttt{for}\ \{\ qs_1;\ z \leftarrow \texttt{equiJoin}(x \Rightarrow k_x, y \Rightarrow k_y)(xs, ys);\ qs_2';\ qs_3';\ qs_4'\ \}\ \texttt{yield}\ let'}$$

$$\textsc{com-cross}\quad \frac{\begin{array}{c} x \notin \mathcal{R}^*[\![ys]\!] \qquad x \in \mathcal{R}[\![k_x]\!] \qquad y \in \mathcal{R}[\![k_y]\!] \qquad t' = [x := z.\_1][y := z.\_2]t \\ \mathcal{R}[\![k_y]\!] \cap \mathcal{G}[\![qs_1 +\!\!+ qs_2]\!] = \emptyset \qquad \mathcal{R}[\![k_x]\!] \cap \mathcal{G}[\![qs_1 +\!\!+ qs_2]\!] = \emptyset \end{array}}{\texttt{for}\ \{\ qs_1;\ x \leftarrow xs;\ qs_2;\ y \leftarrow ys;\ qs_3\ \}\ \texttt{yield}\ let \mapsto \texttt{for}\ \{\ qs_1;\ z \leftarrow \texttt{cross}(xs, ys);\ qs_2';\ qs_3'\ \}\ \texttt{yield}\ let'}$$

$$\textsc{com-map}\quad \frac{}{\texttt{for}\ \{\ x \leftarrow xs\ \}\ \texttt{yield}\ let \mapsto xs.\texttt{map}(x \Rightarrow let)}$$

Fig. 10. Rules introducing comprehension combinators as part of the COMBINE transformation. In COM-FMAP2, COM-JOIN and COM-CROSS, the value of t in $t' = [x := z.\_1][y := z.\_2]t$ ranges over qs_2, qs_3, qs_4, and let.

`FlinkOps` and `SparkOps` are defined in terms of the corresponding native operators exposed by the backend API. For example, if one can extract the backing Flink `DataSet` of a `FlinkBag` `xs` with a `asDataSet(xs)` call, the `equiJoin` method in `FlinkOps` will look as follows.

```
def equiJoin[A,B,K](kx: A => K, ky: B => K)(xs: Bag[A], ys: Bag[B]): Bag[(A,B)] =
  FlinkBag((asDataSet(xs) join asDataSet(ys)) where kx equalTo ky))
```

To introduce these combinators we use a rule-based comprehension compilation strategy called COMBINE (Figure 10). In addition to COM-JOIN and COM-CROSS, Figure 10 lists rules for each of the three monad operators – `map`, `flatMap`, and `withFilter`. Each rule eliminates at least one qualifier in the matched comprehension and introduces a binary combinator or a monad operator. The `flatMap` rule comes in two flavors – if the eliminated generator variable x is referenced in subsequent terms we use COM-FMAP2, otherwise we use COM-FMAP1.

The rules rely on some auxiliary functions. As before, $\mathcal{R}[\![t]\!]$ denotes the set of symbols referenced by t, $\mathcal{R}^*[\![t]\!]$ transitive closure (i.e., the set of symbols upon which t either directly or indirectly depends), and $\mathcal{G}[\![qs]\!]$ denotes the set of generator symbols bound in the qualifier sequence qs. For example, the premise of COM-FILTER states that p should reference x and not reference generator symbols bound in qs_1 or qs_2.

For the sake of readability, the rules in Figure 10 are slightly simplified. The actual implementation maintains the *Emma Core* form. For example, instead of xs, COM-FILTER will match a let_{xs} term

$$\text{SUFFIX}[\![let_{xs}]\!] = \{\ vals;\ defs;\ xs\ \}$$

and rewrite it using fresh symbols `f` and `ys` as follows.

$$\{\ vals;\ \texttt{val f} = x \Rightarrow p;\ \texttt{val ys} = xs\texttt{.withFilter(f)};\ defs;\ \texttt{ys}\ \}$$

The COMBINE translation iteratively applies the first matching rule. The specific matching order is indicated on Figure 10. It ensures that (*i*) filters are pushed down as much as possible, (*ii*) flattening occurs as early as possible, and (*iii*) the join-tree has left-deep structure. The resulting dataflow graph thereby follows heuristics exploited by rule-based query optimizers [26]. To illustrate the COMBINE rewrite, consider the normalized comprehension from Section 5.3.2.

```
for {
  m <- movies; c <- credits; p <- people
  if m.id == c.movieID; if p.id == c.personID
} yield (m.title, p.name)
```

The COMB-JOIN rule first combines `m` and `c` in a new generator `u` (left), and then `u` and `p` in `v` (right).

```
for {
  u <- LocalOps.equiJoin(
        m => m.id, c => c.movieID
       )(movies, credits)
  p <- people
  if p.id == u._2.personID
} yield (u._1.title, p.name)
```

```
for {
  v <- LocalOps.equiJoin(
         u => u._2.personID, p => p.id
       )(LocalOps.equiJoin(
           m => m.id, c => c.movieID
         )(movies, credits), people)
} yield (v._1._1.title, v._2.name)
```

Finally, COMB-MAP rewrites the resulting single-generator comprehension as a `map` call.

```
LocalOps.equiJoin(u => u._2.personID, p => p.id)(
  LocalOps.equiJoin(m => m.id, c => c.movieID)(
    movies,
    credits),
  people).map(v => (v._1._1.title, v._2.name))
```

The COMBINE translation is complemented by an extension of the `Bag` specialization procedure outlined in Section 6.6. In addition to `Bag` constructors, we also specialize combinator applications, replacing `LocalOps` with either `FlinkOps` or `SparkOps` depending on the selected backend.

7.3 Relational Algebra Specialization in Spark

Using the `Column`-based `select`, `filter`, and `join` operators instead of the lambda-based `map`, `filter` and `join` alternatives in Spark has several advantages. First, `Column` expressions are compiled to programs which operate directly on top of Spark's managed runtime. This allows to avoid the costly object serialization and deserialization overhead induced by the need to convert the input of Scala lambdas between Spark's managed runtime and Scala's object representations. Second,

Column expressions are reflected in the Dataset IR and thereby can be targeted by an ever growing set of optimizations as the Spark ecosystem evolves. To enable these benefits, we extend Spark's optimize pipeline with a corresponding specializing transformation.

The specialization operates in two steps. First, we identify lambdas that can be converted to a Column expression and are used in a specializable dataflow operator. Second, we specialize the matched dataflow operators and lambdas.

To model the set of supported Spark Column expressions we use an Expr ADT and an interpretation eval : Expr ⇒ Column. Lambda specialization is restricted to lambdas without control-flow and preserves the layout of the lambda body. We map over the *vdefs* in the lambda body *let* block and check whether their right-hand-sides can be mapped to a corresponding Expr. If this is true for all *vdefs*, we can specialize the lambda and change its type from A ⇒ B to Expr ⇒ Expr.

To illustrate the rewrite, consider the top-level join of the dataflow from Section 7.2. The u ⇒ u._2.personID lambda is specialized as follows (the *Emma Core* version of the input is on the left and the specialized result on the right).

```
val kuOrig = (u: (Movie, Credit)) => {
  val x1 = u._2
  val x2 = x1.personID
  x2
}
```

```
val kuSpec = (u: Expr) => {
  val x1 = Proj(u, "_2")
  val x2 = Proj(x1, "personID")
  x2
}
```

Since all other lambdas in this example can be specialized in a similar way, we can also specialize the equiJoin and map applications that use them. To that end we define an object SparkNtv with specialized dataflow operators equiJoin, select, and project corresponding to the regular equiJoin, map, and withFilter operators from the Bag and ComprehensionCombinators APIs. For example, the definition of equiJoin looks as follows.

```
def equiJoin[A, B, K](kx: Expr => Expr, ky: Expr => Expr)(xs: Bag[A], ys: Bag[B]) = {
  val (us, vs) = (asDataset(xs), asDataset(ys)) // extract Spark Datasets from Bags
  val cx = eval(kx(Root(us))) // construct Column expresion from kx
  val cy = eval(ky(Root(vs))) // construct Column expresion from ky
  SparkBag(us.joinWith(vs, cx === cy)) // wrap a Spark Dataset join in a new SparkBag
}
```

Expr is an ADT that models the subset of the Column language supported by *Emma*. Root(us) and Root(vs) embed us and vs into the Expr ADT. The enclosing kx and ky calls invoke the specialized key selector functions, which in turn construct Expr values for the us and vs join keys. Finally, the eval calls translate these join keys from Expr to Column values.
The implementation accepts the SparkBag instances xs and ys and the specialized lambdas kx and ky. First, we extract the Dataset instances backing xs and ys. We then use those to evaluate kx and ky, obtaining Column expressions for the corresponding join keys. Finally, we construct a new Spark Dataset using the joinWith operator and wrap the result in a SparkBag instance.

The presented specialization approach ensures that we implement *Emma* dataflows in terms of the more efficient, optimizable Column-based Spark API whenever possible, and in terms of the lambda-based Spark API otherwise. This strategy is also future-proof – when a new Spark version rolls out, we only need to add support for new Column expressions to the lambda specialization logic. To make use of these extensions, clients only need to re-compile their *Emma* code.

8 FOLD FUSION

The FOLD-FUSION optimization presented here resolves the issues outlined in Example 2.3 and is enabled by several *Emma* design aspects. First, the Bag API is based on the UNION-representation as a model for distributed data and fold as a model for parallel collection processing (Section 4.1). Second, the API supports nested computations – the groupBy method transforms a Bag[A] into a Bag of Groups where each group contains a values member of type Bag[A] (Section 5.3.3). Third, quote-delimited embedding allows for reflecting such computations in *Emma Core* (Section 6).

The FOLD-FUSION optimization is defined as FOLD-GROUP-FUSION ∘ FOLD-FOREST-FUSION. In the following, we discuss each of the two rewrites in detail. As a running example, we use a code snippet which computes min and avg values per group from a bag of points grouped by their label.

```
val stats = for (Group(label, pnts) <- points.groupBy(_.label)) yield {
  val poss = for (p <- pnts) yield p.pos
  val min  = stat.min(D)(poss)
  val avg  = stat.avg(D)(poss)
  (label, min, avg)
}
```

8.1 Fold-Forest Fusion

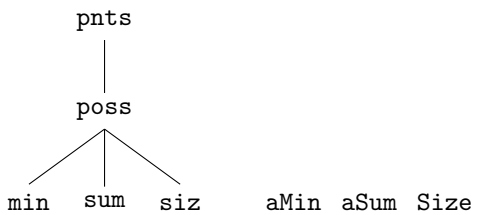
FOLD-FOREST-FUSION rewrites a tree of folds over different UNION-algebras as a single fold over a corresponding tree of UNION-algebras in three steps.

8.1.1 Fold Inlining and Fold-Forest Construction. In the first step, we inline all aliased folds and extract a forest of of fold applications. The roots of the trees in this forest represent distinct Bag instances, leaf nodes represent fold applications, and inner nodes represent linear Bag comprehensions. A *linear comprehension* has the general form (omitting possibly occurring guards)

$$\texttt{for}\ \{\ x_1 \leftarrow let_1\ ;\ \ldots\ ;\ x_n \leftarrow let_n\ \}\ \texttt{yield}\ let_{n+1}$$

where each generator references the symbol bound from the previous one, i.e. $\forall 1 \leq i < n:\ x_i \in \mathcal{R}[\![let_{i+1}]\!]$. The first step in our running example expands the definitions of stat.min and stat.avg (depicted on the left). The forest consists of a single tree rooted at pnts with one inner node – poss – and three leave nodes – min, sum, and siz (depicted on the right).

```
for (Group(label, pnts) <- ...) yield {
  val poss = for (p <- pnts) yield p.pos
  val aMin = stat.Min(D)
  val min  = poss.fold(aMin)
  val aSum = stat.Sum(D)
  val sum  = poss.fold(aSum)
  val siz  = poss.fold(Size)
  val avg  = sum / siz
  (label, min, avg)
}
```



We then collapse each three in a bottom-up way by a FOLD-FOREST-FUSION rewrite, realized as interleaved application of two rewrite rules. The BANANA-FUSION rewrite merges leaf siblings into a single leaf, and CATA-FUSION merges a single leaf node with its parent.

8.1.2 Banana-Fusion. This step is enabled by the *banana-split* law [9], which states that a pair of folds can be fused into a single fold over a pair of algebras. In *Emma* terms, the law states that

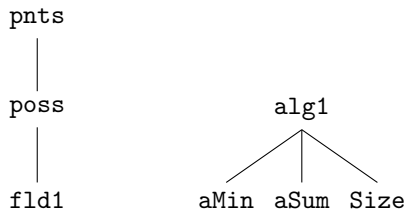
$$(xs.\texttt{fold}(alg_1), xs.\texttt{fold}(alg_2)) = xs.\texttt{fold}(\texttt{Alg2}(alg_1, alg_2))$$

where `Alg2` represents the fusion of two algebras and is defined as follows.

```
class Alg2[A,B1,B2](a1: Alg[A,B1], a2: Alg[A,B2]) extends Alg[A,(B1,B2)] {
  val zero = (a1.zero, a2.zero)
  val init = (x) => (a1.init(x), a2.init(x))
  val plus = (x, y) => (a1.plus(x._1, y._1), a2.plus(x._2, y._2))
}
```

Banana-split generalizes to n-ary tuples. A single application of the above equation from left to right can therefore "fuse" leafs sharing a common parent. In our running example, we fuse the `aMin`, `aSum`, and `Size` algebras as `alg1` and the corresponding `min`, `sum` and `siz` folds as `fld1`. The three leafs of the original fold tree collapse into a single leaf (on the left), while the original structure is reflected in the tree of algebras (on the right). We use dedicated types `AlgN` to encode the result of `N` banana-fused algebras in a type-safe manner.

```
val poss = for (p <- pnts) yield p.pos
...
val alg1 = Alg3(aMin, aSum, Size)
val fld1 = poss.fold(alg1)
val min  = fld1._1
val sum  = fld1._2
val siz  = fld1._3
...
```



8.1.3 Cata-Fusion. This rewrite is inspired by the *cata-fusion* law [9]. It states that a fold over a recursive datatype (also called a *catamorphism*) can be fused with a preceding `map`f application.

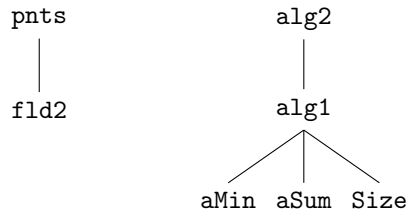
$$xs.\texttt{map}(f).\texttt{fold}(a) = xs.\texttt{fold}(\texttt{AlgMap}(f, a))$$

The `AlgMap` algebra fuses the per-element application of f with a child algebra a.

```
class AlgMap[A,B,C](f: A => B, a: Alg[B,C]) extends Alg[A,C] {
  val zero = a.zero
  val init = x => a.init(f(x))
  val plus = a.plus
}
```

The `poss` definition in our running example is a `Bag` comprehension with a single generator and no guards, so due to the DESUGAR$_{\texttt{Bag}}$ scheme it is equivalent to a `map` call. We therefore can apply the *cata-fusion* law directly and fuse `poss` with `fld1`. Observe the symmetry between the original tree of folds and the resulting tree of algebras in the final result.

```
...
val alg1 = Alg3(aMin, aSum, Size)
val alg2 = AlgMap(p => p.pos, alg1)
val fld2 = pnts.fold(alg2)
val min  = fld2._1
val sum  = fld2._2
val siz  = fld2._3
...
```



Since all Bag comprehensions admit a catamorphic interpretation [33] we can fuse arbitrary linear comprehensions using two additional types of fused algebras. Folds ys.fold(a) where ys is a comprehension of the form

$$\texttt{val}\ ys = \texttt{for}\ \{\ x \leftarrow xs\ ;\ \texttt{if}\ let_1\ ;\ldots;\ \texttt{if}\ let_n\ \}\ \texttt{yield}\ \{\ x\ \}$$

are fused as xs.fold(AlgFilter(p, a)), where AlgFilter is defined as

```
class AlgFilter[A,B](p: A => Boolean, a: Alg[A,B]) extends Alg[A,B] {
  val zero = a.zero
  val init = x => if (p(x)) a.init(x) else a.zero
  val plus = a.plus
}
```

and the predicate p is constructed as a conjunction of the let_i guards.

$$\texttt{val}\ p = x \Rightarrow let_1\ \&\&\ \ldots\ \&\&\ let_n$$

Similarly, folds ys.fold(a) where ys is a linear comprehension of the general form

$$\texttt{for}\ \{\ x_1 \leftarrow \{\ xs\ \}\ ; x_2 \leftarrow let_2; \ldots;\ x_n \leftarrow let_n\ \}\ \texttt{yield}\ let_{n+1}$$

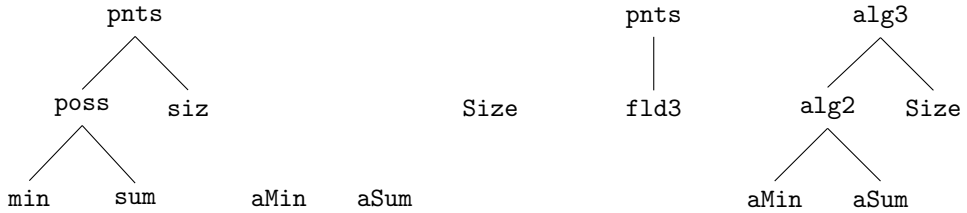
are fused as xs.fold(AlgFlatMap(f, a)). The AlgFlatMap is thereby defined as

```
class AlgFlatMap[A,B,C](f: A => Bag[B], a: Alg[B,C]) extends Alg[A,C] {
  val zero = a.zero
  val init = x => f(x).fold(a)
  val plus = a.plus
}
```

and the argument f is constructed as follows.

$$\texttt{val}\ f = x_1 \Rightarrow \texttt{for}\ \{\ x_2 \leftarrow let_2\ ;\ldots;\ x_n \leftarrow let_n\ \}\ \texttt{yield}\ let_{n+1}$$

With this extension, CATA-FUSION can collapse fold trees with arbitrary shape. To illustrate that, consider a variation of the running example where the siz aggregate is defined not as poss.fold(Size) but directly as pnts.fold(Size). Compared to the original variant, we now fuse only two leafs (aMin and aSum) in the first step, and need an additional BANANA-FUSION between alg2 and Siz in order to construct alg3 at the end. The (left) original and the (right) resulting fold and algebra trees have the following shape.



The CATA-FUSION rewrite rules are closely related to the *foldr/build* rule used by Gill et al. [31] for deforestation of functional collections. In particular, the rule can be generalized to folds of arbitrary ADTs (the presentation in [31] assumes *cons*-style lists), and the CATA-FUSION rules presented above can be recast as instances of this generalized *fold/build* rule for *union*-style bags.

8.2 Fold-Group Fusion

While FOLD-FOREST-FUSION ensures that multiple aggregates derived from the same `Bag` instance can be computed in a single pass, FOLD-GROUP-FUSION fuses group `values` consumed by a single `fold` with a preceding `groupBy` operation that constructs the groups. FOLD-FOREST-FUSION therefore enables a subsequent FOLD-GROUP-FUSION in situations where the group `values` is consumed by multiple folds in the *Emma Source* code. In our running example, we managed to rewrite the tree of folds consuming `pnts` as a single fold consuming a mirrored tree of algebras.

```
val ptgrs = points.groupBy(_.label)
val stats = for (Group(label, pnts) <- ptgrs) yield {
  ... // constructs the tree of algebras rooted at alg2
  val fld2 = pnts.fold(alg2)
  ... // projects min, max, siz aggregates from fld2 and computes avg
  (label, min, avg)
}
```

FOLD-GROUP-FUSION will match the `ptgrs` `groupBy` application, as it is used only once and this use occurs in the right-hand-side of a generator in the `stats` comprehension. The rewrite is subject to two conditions. First, the `values` field bound from each group (`pnts`) must be used only once and this use should be as a target of a `fold` application. Second, the algebra passed to this `fold` application (`alg2`) should not depend on other values bound by the enclosing comprehension (such as `label`). Since both conditions are met, FOLD-GROUP-FUSION pulls the *vdefs* constructing `alg2` out of the `stats` comprehension and redefines `ptgrs` as a `foldGroup` call.

```
... // constructs the tree of algebras rooted at alg2
val ptgrs = LocalOps.foldGroup(_.label, alg2)
val stats = for (Group(label, fld2) <- ptgrs) yield {
  ... // projects min, max, siz aggregates from fld2 and computes avg
  (label, min, avg)
}
```

Similar to the combinators introduced in Section 7, `foldGroup` is defined in a `RuntimeOps` trait and mixed into `LocalOps`, `SparkOps`, and `FlinkOps`. The subsequent backend specialization replaces `LocalOps` with one of the other two implementations and enables targeting the right primitives in the parallel dataflow API. For example, `SparkOps` defines `foldGroup` in terms of the underlying `RDD` representation `us` as follows.

```
def foldGroup[A,B,K](xs: Bag[A], k: A => K, a: Alg[A,B]): Bag[Group[K,B]] =
```

```
xs match {
  case SparkBag(us) => SparkBag(us
    .map(x => k(x) -> a.init(x)) // prepare partial aggregates
    .reduceByKey(a.plus) // reduce by key
    .map(x => Group(x._1, x._2))) // wrap the result (k,v) pair in a group
}
```

The fused version listed above is more efficient than the original version from the beginning of this section. Instead of shuffling all elements of `xs`, it pushes the application of `a.init` and `a.plus` to the `xs` partitions and shuffles at most one value per key and partition. As the number of distinct grouping keys typically is orders of magnitude less than the number of elements in `xs`, this allows to substantially reduce data transfer cost compared to the original version which shuffles `xs` completely.

9 CACHING

The next optimization we propose is automatic `cache` call insertion. As a consequence of the type-based embedding strategy, the distributed collection types exposed by Spark and Flink APIs are *lazy*. The same applies for *Emma*-based `FlinkBag` and `SparkBag` terms, as they are backed by Flink and Spark collections. To illustrate the issues arising from this, consider a more specific, *Emma*-based variation of the second code fragment from Example 2.4.

```
val points = for (d <- Bag.readCSV(/* read text corpus */)) yield
  LPoint(d.id, langs(d.lang), encode.freq(N)(tokenize(d.content)))
val kfolds = kfold.split(K)(points)
var models = Array.ofDim[DVector](K)
var scores = Array.ofDim[Double](K)
for (k <- 0 until K) { // run k-fold cross-validation
  models(i) = linreg.train(logistic, kfold.except(k)(kfolds))
  scores(i) = eval.f1score(models(i), kfold.select(k)(kfolds))
}
...
```

We read a text corpus and convert it into a `Bag` of labeled `points`. Feature extraction is based on tokenizing the document contents into a "bag of words" and feature hashing the resulting representation with the `encode.freq` function. The constructed `points` are then assigned randomly to one of `K` folds. The result is used for `k`-fold cross-validation with a logistic regression model. The `for`-loop implementing the cross-validation thereby accesses the `kfolds` bag in each iteration. If the code is enclosed in an *Emma* quotation, `kfolds` will be specialized as a `SparkBag` or a `FlinkBag`. The uses of `kfolds` in the `train` and `f1score` calls will consequently re-evaluate the backing distributed collection in each of the `K` iterations.

The code fragments below provide two more examples where caching is required.

```
val docs = /* read corpus */
val size = docs.size
val tags =
  if (lang == "de")
    docs.map(posTagger1)
  else if(lang == "fr")
    docs.map(posTagger2)
  else
    docs.map(posTagger3)
val rslts = tags.withFilter(p)
```

```
val edges = Bag.readCSV(/* ... */)
var paths = edges.map(edge2path)
for (i <- 1 until N) {
  paths = for {
    p <- paths
    e <- edges
    if p.head == e.dst
  } yield e.src +: p
}
...
```

In the left example, we read a text corpus `docs`, compute its `size`, and depending on the variable `lang` apply a specific part-of-speech tagger in order to compute `tags`. Since `docs` is referenced more than once, it makes sense to cache it. However, `cache` call insertion should not be too aggressive. Even if we exclude `size` from the snippet, `docs` will still be referenced more than once. However, in this case it is more beneficial to avoid caching in order to pipeline `docs`, `tags` and `rslts` in a single operator chain. To capture these cases, `docs` references in mutually exclusive control-flow blocks should be counted only once.

The right example starts with a bag of `edges` and computes all `paths` of length `N`. Here, caching the loop-invariant `edges` is not too beneficial, as it will only amortize the cost of a single `readCSV` execution per loop. However, the loop-dependent bag `paths` will iteratively construct a dataflow with depth proportional to the value of the loop variable `i`. After the loop, `paths` wraps a dataflow with `N` joins and `N` maps. To assert a constant bound on the size of dataflows updated in a loop iteration we therefore should cache loop-dependent `Bag` instances such as `paths`.

The ADD-CACHE-CALLS optimization covers all cases outlined above based on analysis of the *Emma Core* representation of the optimized program. More precisely, ADD-CACHE-CALLS caches `Bag` instances x if one of the following conditions is met:

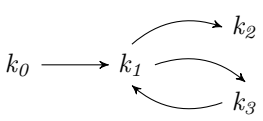
(C1) x is referenced inside a subsequent loop;
(C2) x is referenced more than once in a subsequent acyclic code path;
(C3) x is updated inside a loop.

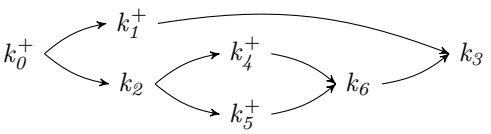
C1 corresponds to situations where (i) x is a value symbol referenced in a continuation k; (ii) k is part of a cycle $k_1, \ldots, k_n$ embedded in the derived control-flow graph; (iii) x is not defined in any of the k_i contained in the cycle.

Let $uses_k(x)$ denote the number of uses for a symbol x in the continuation k (excluding uses in continuation calls), and let $dom(k)$ denote the set of continuations dominated by or equal to k (i.e. all continuation definitions nested in k). **C2** then corresponds to situations where (i) x is a value defined in a continuation k; (ii) the control-flow graph restricted to $dom(k)$ contains at least two strongly connected components S such that $\Sigma_{k \in S} uses_k(x) > 0$; (iii) at least two of these components are also weakly connected between each other.

C3 corresponds to situations where (i) x is a parameter of a continuation definition k, and (ii) the transitive closure of the control-flow graph restricted to $dom(k)$ contains the edge (k, k).

The following control-flow graphs illustrate the three types of checks presented above.





```
val edges = Bag.readCSV(/* ... */)
var paths = edges.map(edge2path)
val it = (0 to 5).toIterator
var i  = null.asInstanceOf[Int]
while(it.hasNext) {


  i = it.next()
  paths = for {
    p <- paths
    e <- edges
    if p.head == e.dst
  } yield e.src +: p




}
... // suffix
```

```
val edges = ANF{ Bag.readCSV(/*...*/) }
val p$1 = ANF{ edges.map(edge2path) }
val it  = ANF{ (0 to 5).toIterator }
val i$1 = null.asInstanceOf[Int]
def k1(i: Int, paths: Bag[Path]) = {
  val hasNext = it.hasNext
  def k3() = { // loop body
    val i$2 = it.next()
    val p$2 = for {
      p <- { paths }
      e <- { edges }
      if ANF{ p.head == e.dst }
    } yield ANF{ e.src +: p }
    k1(i$2, p$2)
  }
  def k2() = ... // suffix
  if (hasNext) k3() else k2()
}
k1(i$1, p$1)
```

Fig. 11. *Emma Source* (left) and *Emma Core* (right) versions of a code snippet that can specialized to Flink's `iterate` operator. For readability, not all terms in the *Emma Core* representation are translated to ANF form (indicated with a surrounding `ANF{...}` call).

The left graph corresponds to the code fragments associated with **C1**[8] and **C3**, where `for` loops are desugared as `while` loops. The loop is represented by k_1 and its body by k_3. **C1** matches for the first fragment, as (*i*) `kfolds` is referenced in the k_3 continuation; (*ii*) k_3 is part of the k_1, k_3 cycle; (*iii*) `kfolds` is defined in $k_0 \notin \{k_1, k_3\}$. **C3** matches for the third fragment, as (*i*) the DSCF-WDO rule converts the `paths` variable to a k_1 continuation parameter, and (*ii*) the closure of the graph restricted to $dom(k_1) = \{k_1, k_3\}$ contains (k_1, k_1).

The right graph corresponds to the part-of-speech tagging code fragment associated with **C2**. The superscript notation k_i^+ indicates that the `docs` value is referenced from the k_i continuation. **C2** matches for this fragment, as (*i*) `docs` is defined in k_0; (*ii*) the graph (without restrictions, as $dom(k_0) = \{k_0, \ldots, k_6\}$) has no cycles, so each continuation k_i represents a trivial strongly connected component S_i, and from those only S_0, S_1, S_4 and S_5 reference `docs`; (*iii*) from the four candidate components, S_0 is weakly connected with S_1, S_4 and S_5. However, if we remove the `size` definition from k_0, condition (*iii*) is not met because S_0 no longer references `docs` and no pair from the remaining $\{S_1, S_4, S_5\}$ is weakly connected.

Qualifying `Bag` instances are cached with a `LocalOps.cache` call. Depending on the enclosing quote, upon backend specialization `LocalOps` is replaced either by `FlinkOps` or `SparkOps`.

10 NATIVE ITERATION SPECIALIZATION IN FLINK

Finally, we propose SPECIALIZE-LOOPS – a transformation which specializes *Emma Core* loops as native Flink iterations. Recall that, in contrast to Spark, Flink lacks full-fledged support for multi-dataflow applications. If the driver application wants to execute multiple dataflows, it has to

[8]For simplicity of presentation, we assume that the `train` and `f1score` calls inside the loop are not inlined.

manually simulate caching of intermediate results, e.g. by writing them to disc. To compensate for this limitation, Flink offers a dedicated `iterate` operator for a restricted class of iterative programs. The transformation described in this section identifies *Core* language patterns corresponding to this class of programs and rewrites them in terms of an `iterate` operator backed by Flink.

As a running example, consider again the `edges` and `paths` code fragment from Section 9. The *Emma Source* and *Core* representations for this example are depicted in Figure 11. In order to be executable as a Flink native iteration, the *Core* program should meet the following criteria:

- `k1` through `k3` should form a control-flow graph corresponding to a simple `while` loop;
- `k1` should have two parameters – an induction variable (`i : Int`) and a bag (`paths : Bag[A]`);
- the induction variable should bind to values in the $[0, N)$ range (in the example $N = 5$);
- with exception of the induction variable update (`i$2`), all *vdefs* in the continuation representing the loop body (`k3`), should form a dataflow graph rooted at the value binding to the bag parameter in the `k1` continuation call (`p$2` in the running example);

If these conditions are met, we can replace the `k1` through `k3` loop with an `iterate` call. The rewrite

- eliminates the `k1` subtree and all preceding values contributing only to the induction variable (in the running example, it will eliminate `it` and `i$1`);
- wraps the original body (minus the induction update *vdef*) in a fresh lambda function (`f$1`);
- rewrites the bag parameter (`paths`) as a *vdef* that binds the result of an `iterate` call;
- appends the body of the original suffix continuation `k2` to the modified root continuation.

The resulting program in our running example looks as follows.

```
val edges = ANF{ Bag.readCSV(/*...*/) }
val p$1 = ANF{ edges.map(edge2path) }
val f$1 = (paths: Bag[Path]) => {
  val p$2 = for {
    p <- { paths }
    e <- { edges }
    if ANF{ p.head == e.dst }
  } yield ANF{ e.src +: p }
  p$2
}
val paths = FlinkNtv.iterate(5)(f$1)(p$1)
... // suffix
```

The `iterate` method is defined in a `FlinkNtv` module and delegates to Flink's native `iterate` operator. Recall that in this case Flink's optimizer can analyze the body and automatically cache loop-invariant data, as `iterate` calls are reflected in the Flink IR (Example 2.4). To avoid the naïve *Emma*-based caching of loop-invariant `Bag` instances, SPECIALIZE-LOOPS precedes ADD-CACHE-CALLS.

11 EVALUATION

To assess the benefits of the proposed *Emma* optimizations we conducted a set of experiments on an on-premise cluster consisting of a dedicated master and 8 worker nodes. Each worker was equipped with two AMD Opteron 6128 CPUs (a total of 16 cores running at 2.0 GHz), 32 GiB of RAM, and an Intel 82576 gigabit Ethernet adapter. All machines were connected with a Cisco 2960S switch. As backends we used Spark 2.2.0 and Flink 1.4.0 – the latest versions to the date of execution. Each backend was configured to allocate 18 GiB of heap memory per worker and reserve 50% of

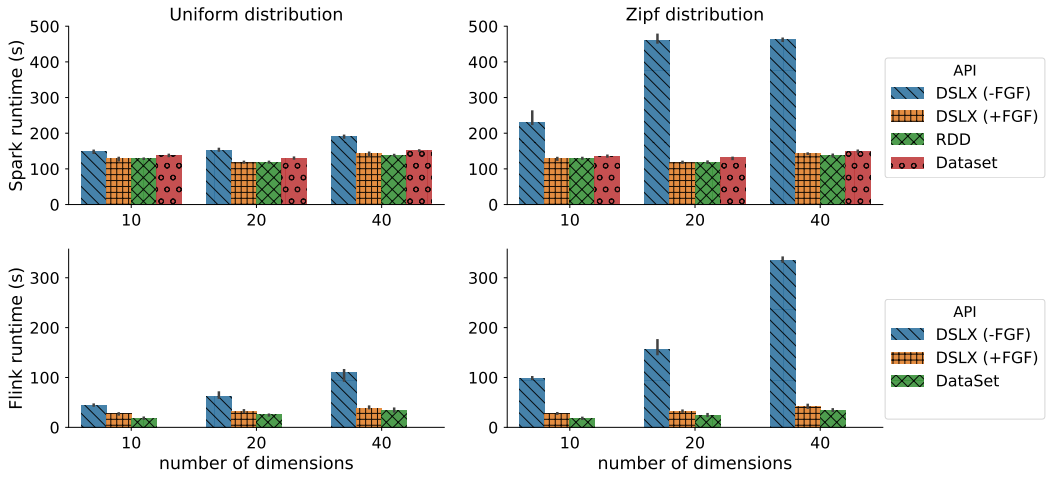


Fig. 12. Effects of fold-group fusion (FGF) in Flink and Spark.

this memory for its managed runtime. Input and output data was stored in an HDFS 2.7.1 instance that was running on the same set of nodes.

The experiments discussed in Section 11.1 through Section 11.4 were executed five times. The associated bar charts in Figure 12 through Figure 15 indicate the median run and the error bars denote the second fastest and second slowest runs. The experiments discussed in Section 11.5 were executed three times and the bars in Figure 16 indicate the median run.

11.1 Fold-Group Fusion

We first assess the effects of fold-group fusion (FGF).

The workload is a single iteration of the *k-means* clustering algorithm [25], using synthetic datasets consisting of points sampled from one of k multivariate Gaussian distributions as input. We used both uniform and Zipf distribution on each of the two backends, resulting in four experiments in total. In each experiment, we scaled the number of data points dimensions from 10 to 40 in a geometric progression, comparing the runtime of two *Emma*-based implementations with FOLD-GROUP-FUSION turned off (-FGF) and on (+FGF). We used a `DataSet` implementation for Flink and `Dataset` and `RDD` implementations for Spark as a baseline.

The results of the four experiments are presented in Figure 12. In the *Emma (-FGF)* version, the k means are computed naïvely with a `map` ∘ `groupBy` chain in Spark and a `reduceGroup` ∘ `groupBy` operator chain in Flink. All points associated with a same centroid must be therefore shuffled to a single machine where their mean is then computed, and the overall runtime is dominated by the size of the largest group. On the other hand, with FGF enabled the sum and count of all points associated with the same centroid are computed in parallel, using a `reduceByKey` operator in Spark and a `reduce` ∘ `groupBy` operator chain in Flink. The associated shuffle step needs to transfer only one partial result per group and per worker, and the total runtime does not depend on the group size. The overall effect is illustrated by the experiment results. Irrespective of the backend, the runtime of the *Emma (-FGF)* implementation grows as we increase the dimensionality of the data. The runtime of the *Emma (+FGF)* and the three baseline variants, however, is not affected by the underlying centroid distribution and only slightly influenced by changes in data dimensionality. The code generated by *Emma (+FGF)* performs on par with the code written directly against the

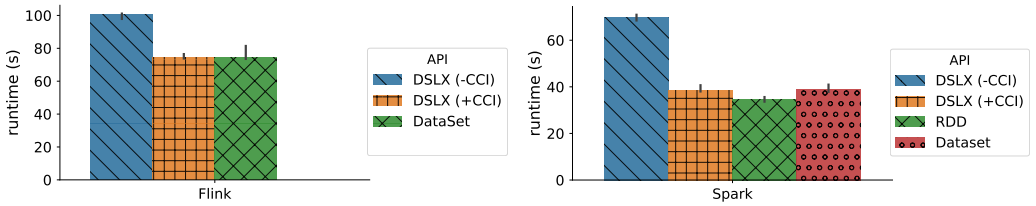


Fig. 13. Effects of cache-call insertion (CCI) in Flink and Spark.

Flink and Spark APIs. The speedup of *Emma (+FGF)* with respect to *Emma (-FGF)* varies. For the uniform distribution, it ranges from 1.58x to 2.82x (Flink) and from 1.16x to 1.35x (Spark). For the Zipf distribution, it ranges from 3.46x to 8.11x (Flink) and from 1.17x to 3.24x (Spark). The effect is stronger if the centroid distribution is skewed, as this skew is reflected in the relative cardinality of the aggregated groups and the total runtime is dominated by the size of the largest group.

11.2 Cache-Call Insertion

The second experiment demonstrates the benefits of the cache-call insertion (CCI) optimization.

The input data was derived from a snapshot of the Internet Movie Database (IMDb)[9] which we subsequently parsed and saved as structured collections of JSON objects. The workload operates in two steps. In the first step, we perform a three-way join between movies, countries, and technical information, selecting information about German titles released in the 1990s and categorized as "motion picture". In the second step, we retrieve six subsets of these titles based on different filter criteria (e.g., titles shot on an Arri film camera or titles with aspect ratio 16:9) and collect each of the six results on the workload driver. The collection obtained after the first step is cached in the *Emma (+CCI)* and the baseline variants, and re-evaluated in the *Emma (-CCI)* variant.

The experiment results are depicted on Figure 13. As in the previous experiment, the optimized *Emma* version is comparable with the baseline implementations. The optimized variant achieves a speedup of 1.35x for Flink and 1.81x for Spark compared to *Emma (-CCI)*. The difference is explained by the underlying caching mechanism. Spark has first-class support for caching and keeps cached collections directly in memory, while Flink lacks this feature. Consequently, the `FlinkOps.cache` primitive inserted by the *Emma* compiler simply writes the cached distributed collection to HDFS. Reads of cached collections therefore are more expensive in Flink and cancel out a fraction of the performance benefit gained by caching.

11.3 Specializing Relational Algebra Operators in Spark

Next, we investigate the benefits of the specializing program transformation from Section 7.3 that replaces `RDD`-based `map`, `filter`, and `join` operators with more efficient `Dataset`-based operators.

The experiments are also based on the IMDb shapshot. To quantify the performance improvement of relational algebra specialization (RAS) we use two different workloads. The first workload ('gender-year-credits') represents a simple three-way join where people and movies are connected via credits with credit type 'actor', emitting pairs of (*person-gender*, *movie-year*) values. The 'sharing-roles' workload looks for pairs of actors who have played the same character in two different movies and starred in a third movie together. For example, Michael Caine (in "Sherlock Holmes, Without a Clue") and Roger Moore (in "Sherlock Holmes in New York") have both played Sherlock

[9] ftp://ftp.fu-berlin.de/pub/misc/movies/database/frozendata/

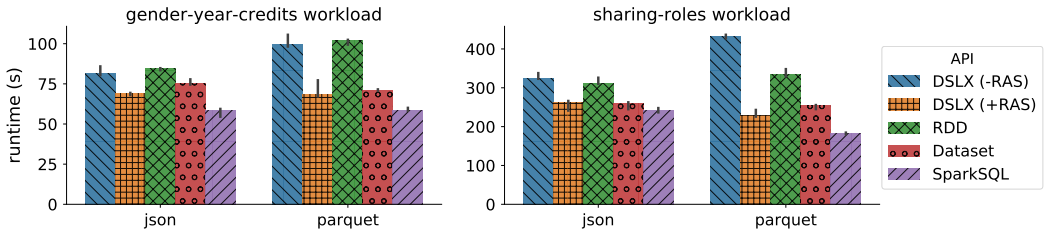


Fig. 14. Effects of relational algebra specialization (RAS) in Spark.

Holmes and acted together in "New York, Bullseye!". We add a Spark SQL baseline implementation and a more efficient columnar format (Parquet) next to the string-based JSON input.

Figure 14 depicts the results for the two workloads. In all experiments, *Emma (-RAS)* performs on par with the `RDD` baseline, and *Emma (+RAS)* is comparable with the `Dataset` implementation. Notably, for Parquet files we observe a higher *(+RAS)* to *(-RAS)* speedup (1.46x and 1.90x) compared to the 1.18x and 1.23x observed for JSON. This difference is explained by the more aggressive optimizations performed by Spark in the first case. `Dataset` dataflows reading Parquet data can utilize Parquet's columnar format and push `select` and `project` operators directly to the Parquet reader. In *Emma*, local predicates are pushed on top of the base collections as a result of the COMBINE translation scheme from Figure 10, and the following RAS enables selection push-down for Parquet. However, the COMBINE scheme currently does not automatically insert projections, whereas the Spark SQL implementation enables both selection and projection push-down. Consequently, *Emma (+RAS)* variants for Parquet are 1.17x and 1.25x slower than the Spark SQL baseline. The COMBINE translation scheme can be extended with a suitable projection rule to narrow this gap.

11.4 Native Iteration Specialization

Finally, we investigate the effects of the Flink-specific native iterations specialization (NIS).

The NIS experiment is also based on the IMDb snapshot. The workload first selects ID pairs for directors billed for the same movie, considering only titles released between 1990 and 2010. The result is treated as a bag of edges, and a subsequent iterative dataflow computes the first five steps of the connected components algorithm proposed by Ewen et al. in [20], using the FIXPOINT-CC variant from Table 1 in [20]. The algorithm initializes each vertex with its own component ID. In every iteration, each vertex first sends a message with its current component ID to all its neighbors, and then sets its own component ID to the minimum value of all received messages.

The results can be seen on Figure 15. Note that the *Emma (-NIS)* variant performs CCI, so the loop-independent collection of edges and the component assignments at the end of each iteration are saved to HDFS by the inserted `FlinkOps.cache` calls. CCI is not needed for the *Emma (+NIS)* variant, as in this case the Flink runtime manages the iteration state and loop-invariant dataflows in memory. Overall, the *Emma (+NIS)* variant and the baseline `DataSet` implementation are 4x faster compared to the *Emma (-NIS)*.

11.5 Cumulative Effects

To conclude, we investigate the cumulative effects of all optimizations using an end-to-end data analytics pipeline.

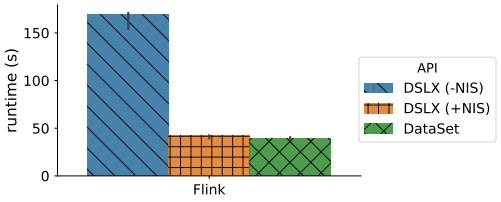


Fig. 15. Effects of native iterations specialization (NIS) in Flink.

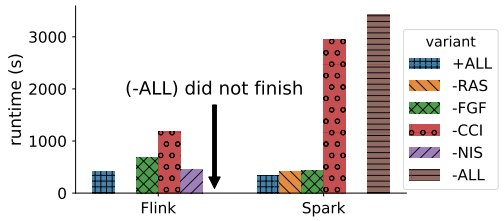


Fig. 16. Cumulative optimization effects for the NOMAD use case.

The workload for this experiment is based on data obtained from the NOMAD repository[10] – an archive of output data from computer simulations for material science in a common hierarchical format [28]. For the purposes of our experiment, we downloaded the complete NOMAD archive and normalized the original hierarchical structure as a set of CSV files. The normalized result contains data about (1) simulated physical systems and (2) positions of the simulated atoms, as well as meta-data about (3) periodic dimensions and (4) simulation cells.

The workload pipeline is structured as follows. We first join the data from the four CSV sources listed above and apply a Radial Distribution Function (RDF) conversion. This yields a collection of dense vectors characterizing the result of each simulation. We then execute n runs of the first m iterations of a k-means clustering algorithm, keeping track of the best solution obtained. At the end of the pipeline this solution is saved to HDFS. To obtain sufficiently small numbers for a single experiment run, for the purposes of the experiment we used $n = 2$, $m = 2$ and $k = 3$. The values for n and m will likely be higher in practice.

The workload is encoded as an *Emma* program and compiled in six different variants for each of the two supported backends. The name *-ALL* (*+ALL*) denotes a variant where all optimizations are disabled (enabled), and *-OPT* a variant where all optimizations except *OPT* are enabled.

The results are depicted on Figure 16. Spark runtimes vary between 346s for the *+ALL* and 3421s for *-ALL*, while in Flink *+ALL* achieves 413s and *-CCI* is slowest with 1186s (the *-ALL* variant did not finish successfully). For both backends, the largest penalty is for a disabled *CCI* optimization – 2.87x for Flink and 8.5x for Spark. With disabled *FGF*, the slowdown is 1.27x for Spark and 1.67x for Flink. Disabling *NIS* results in 1.09x slowdown for Flink, and *-RAS* in 1.22x slowdown for Spark.

The observed results suggest that the most important optimization is CCI. We believe that this is typical for all data analytics pipelines where feature conversion and vectorization is handled by a CPU-intensive computation in a `map` operator. In such pipelines, feature conversion is the last step before an iterative part of the program that performs cross-validation, grid-search, an iterative ML method, or a nested combination of those. If the collection of feature vectors is not cached, the feature conversion `map` is re-computed for each inner iteration. For example, in the NOMAD pipeline this results in $n * m = 4$ repeated computations.

12 RELATED WORK

In this section we review related work. Section 12.1 discusses related applications of the formal foundations in this paper, while Section 12.2 discusses related DSLs.

[10] https://nomad-repository.eu/

12.1 Formal Foundations

Using monads to structure and reason about computer programs dates back to Moggi [51], who suggests them as a referentially transparent framework for modeling effectful computations. Comprehension syntax was first introduced by Wadler [66, 67]. Employing comprehensions in order to systematically define database query languages for different bulk types can be traced back to the work of Trinder [18, 62, 63]. Notably, following unpublished work by Wadler [65], Trinder's work is based on *ringads* – monads extended with functions *zero* and *combine*. While Trinder's ringads require only that *zero* is a unit of *combine*, if we add associativity and commutativity we end with the Union-style bag and monad used in this paper.

Buneman and Tannen advocate that query languages should be constructed from the primitive notion of set catamorphisms [60]. They show that existing set-valued query languages can be formalized based on that notion and generalized to other collection types such as lists and bags [13]. As a proof-of-concept, they propose Comprehension Language (CL) – an external DSL based on comprehensions [12]. Notably, the IR proposed for CL does not rely on monads. Instead, comprehension syntax is defined directly in terms of catamorphisms on Union-style collections.

Similarly, Fegaras employs monoids as a basic notion and proposes a core calculus that defines comprehension syntax in terms of monoid catamorphisms [21]. Fegaras and Mayer show that this calculus can be used to formalize Object Query Language (OQL) – a standardized language for object-oriented Database Management Systems (DBMSs) [24].

Despite some differences in naming and notation, the formal development suggested in these two lines of work is quite similar. The free monoids used in [21] and the collection types associated with *sr_comb* in [13] coincide. Similarly, *hom* (Definition 5 in [21]) / *sr_comb* (Section 2.2 in [13]) define structural recursion schemes over a free monoid / collection type. In the case of a free commutative monoid in [21] and the bag collection type in [13], the corresponding *hom* / *sr_comb* definition coincides with the *fold* definition from Section 4.1.

The application of Union-style bags and monads in this paper is based on the work of Grust [33, 36]. Similar to both Buneman and Fegaras, Grust starts from the basic notion of catamorphisms, but differs in the following aspects. First, he relies on collections in insert representation (the union representation is briefly presented in [33]). Second, he explicitly derives a monad with zero from the initial algebra of the underlying collection type, and defines comprehension syntax in terms of this monad similar to the work of Wadler. In contrast to the monad comprehension scheme from [66], however, the one given by Grust supports generators that range over multiple collection types, employing an implicit type coercion approach similar to the one proposed by Fegaras in [21]. Third, Grust proposes that comprehensions are a useful representation for defining and reasoning about optimizing program transformations. Finally, he also suggests a compilation strategy based on rule-based elimination of comprehensions using comprehension combinators.

We follow Grust in all but the first aspect, where like Buneman and Fegaras we opt for the union representation. Our choice is motivated by the distributed nature of the underlying execution platforms. The connection between Bag-Union and its associated structural recursion scheme Bag-Fold for the design of parallel programs has already been highlighted by Skillicorn [56, 57] and Steele [43]. Our contribution in that regard is to highlight the relevance of this methodology for the design of APIs and DSLs that target parallel dataflow engines. We also extend a comprehension-based IR such as *Emma Core* with full-fledged support for control-flow, filling the semantic gap between previous work and typical data analytics use-cases.

Recently, Gibbons brought back attention to [65] in a survey article [29], arguing that ringads and ringad comprehensions form a better foundation and query language than monads. Although we don't follow the ringad nomenclature, our work obviously supports this claim. We also highlight the

connection between the associativity and commutativity laws in the underlying ringad definition and data-parallel execution.

12.2 Related DSLs

Related DSLs can be categorized in a two-dimensional space. The first dimension denotes the implementation strategy in accordance with the classification scheme from Figure 1. The second dimension classifies DSLs based to their execution backend – a parallel dataflow engine, an RDBMS, or a custom runtime. We review DSLs which coincide with *Emma* in at least one of these dimensions.

12.2.1 External DSL Targeting Parallel Dataflow Engines. Pig [53] and Jaql [8] are external scripting DSLs compiled to a cascade of Hadoop MapReduce jobs. Hive [61] provides warehousing capabilities on top of Hadoop or Spark using a SQL-like DSL. SparkSQL [6] is the default SQL layer implemented on top of Spark. SCOPE [69] is a SQL-like DSL developed by Microsoft which runs on a modified version of the Dryad dataflow engine [40]. External DSLs as the ones mentioned above provide automatic optimization (such as join order optimization and algorithm selection) at the cost of more limited expressive power. In particular, they do not treat UDFs as first-class citizens and lack first-class support for control-flow. Optimizations related to these language aspects therefore are designed in an ad-hoc manner. For example, PeriSCOPE [37] optimizes SCOPE UDFs, but relies on Cecil[11] for code inspection and code synthesis and ILSpy[12] for bytecode decompilation. The *Emma Core* IR presented in this paper integrates both UDFs and control-flow as first-class citizens. This presents a unified methodological framework for defining and reasoning about optimizations related to these constructs. At the same time, optimizations traditionally associated with SQL can be integrated on top of the proposed IR based on the first-class comprehension syntax.

12.2.2 Embedded DSLs Targeting RDBMS Engines. The most popular example of an EDSL that targets RDBMS engines is Microsoft's LINQ [49]. Database-Supported Haskell (DSH) [32] enables database-supported execution of Haskell programs through the Ferry programming language [34]. As with external DSLs, the main difference between *Emma* and these languages is the scope of their syntax and IR. LINQ's syntax and IR are based on chaining of methods defined by an `IQueryable` interface, while DSH is based on Haskell list comprehensions desugared by the method suggested by Jones and Wadler [41]. Neither of the two EDSLs lifts control-flow constructs from the host language in its respective IRs. In addition, because they target RDBMS engines, they also restrict the set of host language expressions that can be used in selection and projection clauses to a subset that can be mapped to SQL. In contrast, *Emma* does not enforce such restriction, as host-language UDFs are natively supported by the targeted parallel dataflow engines. Nevertheless, the connection between SQL-based EDSLs and *Emma* deserves further investigation. In particular, transferring avalanche-safety [35, 64] and normalization [16] results obtained in this space to *Emma Core* most likely will further improve the runtime performance of compiled *Emma* programs.

12.2.3 Embedded DSLs Targeting Parallel Dataflow Engines. The Spark and Flink EDSLs and their problems are discussed in detail in Section 2.3. FlumeJava [15] and Cascading[13] provide an API for dataflow graph assembly that abstracts from the underlying engines and ships with a dedicated execution planner. Summingbird [11] and Apache Beam[14] (an open-source descendant of the Dataflow Model proposed by Google [2]) provide a unified API for batch and stream data processing that is also decoupled from the specific execution backend. DSL terms in all of the above

[11] http://www.mono-project.com/Cecil
[12] http://wiki.sharpdevelop.net/ilspy.ashx
[13] https://www.cascading.org/
[14] https://beam.apache.org/

examples are delimited by their type. Consequently, they suffer from the deficiencies associated with the Flink and Spark EDSLs illustrated in Section 2.3.

Jet [1] is a EDSL that supports multiple backends (e.g., Spark, Hadoop) and performs optimizations such as operator fusion and projection insertion. Jet is based on the Lightweight Modular Staging (LMS) framework proposed by Rompf [54], and as such does not suffer from the limited reflection capabilities associated with the type-based EDSLs listed above. Nevertheless, the Jet API is based on a distributed collection (`DColl`) which resembles more Spark's `RDD` than *Emma*'s `Bag` interface. For example, the `DColl` relies on explicit `join` and `cache` operators and lacks optimizations which introduce those automatically.

The Data Intensive Query Language (DIQL) [22] is a SQL-like Scala EDSL. Similar to *Emma*, DIQL is based on monoids and monoid homomorphisms. However, while *Emma* programs are embedded as quoted Scala code, DIQL programs are embedded as quoted string literals (for that reason, DIQL can also be classified as an external DSL with a quote-delimited string embedding in Scala). DIQL programs therefore cannot benefit from the syntactic reuse and tooling of the syntax-sharing approach adopted by *Emma*. On the other side, while *Emma* reuses Scala's `for`-comprehension syntax, DIQL's stand-alone syntax allows for more "comprehensive" comprehension syntax in the sense coined by Wadler and Peyton Jones [41]. Another notable difference between *Emma* and DIQL is their control-flow model. *Emma* supports general-purpose `while` and `do – while` loops, while DIQL relies on a custom `repeat` construct.

An earlier version of *Emma* was presented in prior work [3, 4]. Our original prototype was also based on quotation-based term delineation, exposed an API defined in terms of Union-style bags catamorphisms, and (loosely speaking) supported the Scala syntax formalized in Figure 3. However, the compiler pipeline from [3, 4] and the one presented here differ significantly in their IR. The original design was based on vanilla Scala ASTs and an auxiliary IR layer providing a "comprehension view" over desugared `Bag` monad expressions. This ad-hoc approach severely limited the ability of the DSL compiler to define and execute program transformations in a robust manner. For example, the fold-group-fusion optimization in [3, 4] is sketched only in conjunction with the banana-fusion transformation from Section 8.1.2. A cata-fusion rewrite like the one outlined in Section 8.1.3 was not considered or implemented because of the syntactic variability of the underlying AST terms. The original *Emma* compiler therefore was not able to fuse the tree of folds listed in the beginning of Section 8.1.1, and consequently also not able to apply fold-group-fusion to the enclosing comprehension. On the other side, with *Emma Source* and *Emma Core* this paper provides formal definitions of both the concrete syntax and the IR of the proposed EDSL. A host-language-agnostic, ANF-like IR with first-class support for monad comprehensions offers a basis for the development of robust optimizations that are fully decoupled from the host language IR.

12.2.4 Embedded DSLs with Custom Runtimes. Delite [58] is a compiler framework for the development of data analytics EDSLs that targets heterogeneous parallel hardware. Delite uses an IR based on functional primitives such as *zipWith*, *map* and *reduce*. Delite EDSLs are staged to this IR using an LMS-based stating partial evaluation. From this IR, Delite produces executable kernels which are then scheduled and executed from a purpose-built runtime. A language such as *Emma* can be implemented on top of Delite. To that end, one must (*a*) define *Emma Core* in terms of Delite's IR and (*b*) add support for Flink and Spark kernels to the Delite runtime.

The AL language proposed by Luong et al. [47] is a Scala-based EDSL for unified data analytics. AL programs are translated to a comprehensions-based IR and executed by a dedicated runtime which employs just-in-time (JIT) compilation and parallel `for`-loop generation for IR comprehensions. Similar to AL, we use the monad comprehensions exposed by *Emma Core* as a starting point for

some compiler optimizations. However, *Emma Core* also supports forms of control-flow that cannot be expressed as comprehensions. Similar to DIQL, ALs frontend is based on a quoted strings and suffers from the same syntactic limitations.

13 CONCLUSIONS AND FUTURE WORK

State-of-the-art parallel dataflow engines such as Flink and Spark expose various EDSLs for distributed collection processing, e.g. the `DataSet` DSL in Flink and `RDD` DSL in Spark. We highlighted a number of limitations shared among these EDSLs and identified delimiting based on types as their common root cause. IRs constructed from type-delimited EDSLs can only reflect host language method calls on these types. Consequently, the optimization potential and declarativity attained by type-delimited EDSLs are heavily restricted.

As a solution, we proposed an EDSLs design that delimits DSL terms using quotes. DSLs following this principle can reuse more host-language constructs in their concrete syntax and reflect those in their IR. As a result, quote-delimited EDSLs can attain declarative syntax and optimizations traditionally associated with external DSLs such as SQL.

In support of our claim, we proposed *Emma* – a quote-delimited DSL embedded in Scala. *Emma* targets either Flink or Spark as a co-processor for its distributed collection abstraction. We presented various aspects of the design and implementation of *Emma*. As a formal foundation, in accordance with the operational semantics of the targeted parallel dataflow engines, we promoted bags in union representation and their associated structural recursion scheme and monad. As a syntactic construct, we promoted bag comprehensions using Scala's native `for`-comprehension syntax. As a basis for compilation, we proposed *Emma Core* – an IR that builds on ANF and adds first-class support for monad comprehensions. Demonstrating the utility of *Emma Core*, we developed optimizations solving the issues of state-of-the-art EDSLs identified in the beginning of the paper. Finally, we quantified the performance impact of these optimizations using an extensive experimental evaluation.

The proposed design therefore represents a first step towards reconciling the utility of state-of-the-art EDSLs with the declarativity and optimization potential of external DSLs such as SQL. Nevertheless, in addition to collections, modern data analytics applications increasingly rely on programming abstractions such as data streams and tensors. In current and future work, we therefore plan to extend the *Emma* API with types and APIs reflecting these abstractions. The primary goals in this endeavor are twofold. First, we want to ensure that different APIs can be composed and nested in an orthogonal manner. This means that one can convert a bag into a tensor (composition) or process a stream of tensors (nesting). Second, we want to ensure that the degrees of freedom resulting from this orthogonality do not affect the performance of the compiled program. To achieve that, we will propose optimizations targeting a mix of the available APIs.

A INFERENCE RULES FOR THE ANF AND DSCF TRANSFORMATIONS

$$\textsc{anf-atom}\ \frac{}{a \mapsto \{\ a\ \}} \qquad \textsc{anf-ascr}\ \frac{t \mapsto \{\ ss;\ a\ \}}{t:\ T \mapsto \{\ ss; \texttt{val}\ x = a:\ T;\ x\ \}} \qquad \textsc{anf-fun}\ \frac{t \mapsto t'}{pdefs \Rightarrow t \mapsto pdefs \Rightarrow t'}$$

$$\textsc{anf-if}\ \frac{t_1 \mapsto \{\ ss;\ a\ \} \qquad t_2 \mapsto t_2' \qquad t_3 \mapsto t_3'}{\texttt{if}\ (t_1)\ t_2\ \texttt{else}\ t_3 \mapsto \{\ ss; \texttt{val}\ x = \texttt{if}\ (a)\ t_2'\ \texttt{else}\ t_3';\ x\ \}}$$

$$\textsc{anf-mod}\ \frac{t \mapsto \{\ ss;\ a\ \}}{t.module \mapsto \{\ ss; \texttt{val}\ x = a.module;\ x\ \}} \qquad \textsc{anf-blck}\ \frac{t \mapsto \{\ ss;\ a\ \}}{\{\ t\ \} \mapsto \{\ ss;\ a\ \}}$$

$$\textsc{anf-call}\ \frac{t \mapsto \{\ ss;\ a\ \} \qquad \forall t_{jk}.\ t_{jk} \mapsto \{\ ss_{jk};\ a_{jk}\ \}}{t.m[T_i](t_{jk})_j \mapsto \{\ ss; ss_{1k}; \ldots; ss_{jk}; \ldots; \texttt{val}\ x = a.m[T_i](a_{jk})_j;\ x\ \}}$$

$$\textsc{anf-new}\ \frac{\forall t_{jk}.\ t_{jk} \mapsto \{\ ss_{jk};\ a_{jk}\ \}}{\texttt{new}\ C[T_i](t_{jk})_j \mapsto \{\ ss_{1k}; \ldots; ss_{jk}; \ldots; \texttt{val}\ x = \texttt{new}\ C[T_i](a_{jk})_j;\ x\ \}}$$

$$\textsc{anf-val}\ \frac{t_1 \mapsto \{\ ss_1;\ a_1\ \} \qquad \{\ ss;\ t_2\ \} \mapsto \{\ ss_2;\ a_2\ \}}{\{\ \texttt{val}\ x = t_1; ss;\ t_2\ \} \mapsto \{\ ss_1; [x := a_1]ss_2;\ [x := a_1]a_2\ \}}$$

$$\textsc{anf-var}\ \frac{t_1 \mapsto \{\ ss_1;\ a_1\ \} \qquad \{\ ss;\ t_2\ \} \mapsto \{\ ss_2;\ a_2\ \}}{\{\ \texttt{var}\ x = t_1; ss;\ t_2\ \} \mapsto \{\ ss_1; \texttt{var}\ x = a_1; ss_2;\ a_2\ \}}$$

$$\textsc{anf-asgn}\ \frac{t_1 \mapsto \{\ ss_1;\ a_1\ \} \qquad \{\ ss;\ t_2\ \} \mapsto \{\ ss_2;\ a_2\ \}}{\{\ x = t_1; ss;\ t_2\ \} \mapsto \{\ ss_1; x = a_1; ss_2;\ a_2\ \}}$$

$$\textsc{anf-loop}\ \frac{loop \mapsto loop' \qquad \{\ ss;\ t\ \} \mapsto \{\ ss';\ a\ \}}{\{\ loop; ss;\ t\ \} \mapsto \{\ loop'; ss';\ a\ \}}$$

$$\textsc{anf-wdo}\ \frac{t \mapsto t' \qquad block \mapsto block'}{\texttt{while}\ (t)\ block \mapsto \texttt{while}\ (t')\ block'} \qquad \textsc{anf-dow}\ \frac{t \mapsto t' \qquad block \mapsto block'}{\texttt{do}\ block\ \texttt{while}\ (t) \mapsto \texttt{do}\ block'\ \texttt{while}\ (t')}$$

Fig. 17. Inference rules for the ANF : $Source \Rightarrow Source_{ANF}$ transformation.

$$\textsc{dscf-ref1}\ \frac{x \notin \mathcal{V}}{\mathcal{V} \vdash x \mapsto x} \qquad \textsc{dscf-ref2}\ \frac{\mathcal{V}x = a}{\mathcal{V} \vdash x \mapsto a} \qquad \textsc{dscf-ascr}\ \frac{\mathcal{V} \vdash a \mapsto a'}{\mathcal{V} \vdash a : T \mapsto a' : T} \qquad \textsc{dscf-mod}\ \frac{\mathcal{V} \vdash a \mapsto a'}{\mathcal{V} \vdash a.module \mapsto a'.module}$$

$$\textsc{dscf-fun}\ \frac{\mathcal{V} \vdash block \mapsto let}{\mathcal{V} \vdash pdefs \Rightarrow block \mapsto pdefs \Rightarrow let} \qquad \textsc{dscf-call}\ \frac{\mathcal{V} \vdash a \mapsto a' \qquad \mathcal{V} \vdash \forall a_{jk}.\ a_{jk} \mapsto a'_{jk}}{\mathcal{V} \vdash a.m[T_i](a_{jk})_j \mapsto a'.m[T_i](a'_{jk})_j}$$

$$\textsc{dscf-new}\ \frac{\mathcal{V} \vdash \forall a_{jk}.\ a_{jk} \mapsto a'_{jk}}{\mathcal{V} \vdash \mathsf{new}\ C[T_i](a_{jk})_j \mapsto \mathsf{new}\ C[T_i](a'_{jk})_j} \qquad \textsc{dscf-let}\ \frac{}{\mathcal{V} \vdash \{\ kdefs;\ c\ \} \mapsto \{\ kdefs;\ c\ \}}$$

$$\textsc{dscf-val}\ \frac{\mathcal{V} \vdash b \mapsto b' \qquad \mathcal{V} \vdash \{\ ss;\ c\ \} \mapsto \{\ vdefs;\ kdefs;\ c'\ \}}{\mathcal{V} \vdash \{\ \mathtt{val}\ x = b; ss;\ c\ \} \mapsto \{\ \mathtt{val}\ x = b'; vdefs;\ kdefs;\ c'\ \}}$$

$$\textsc{dscf-var}\ \frac{\mathcal{V} \vdash a \mapsto a' \qquad \mathcal{V}, x \leftarrow a' \vdash \{\ ss;\ c\ \} \mapsto let}{\mathcal{V} \vdash \{\ \mathtt{var}\ x = a; ss;\ c\ \} \mapsto let} \qquad \textsc{dscf-lit}\ \frac{}{\mathcal{V} \vdash lit \mapsto lit}$$

$$\textsc{dscf-asgn}\ \frac{\mathcal{V} \vdash a \mapsto a' \qquad \mathcal{V}, x \leftarrow a' \vdash \{\ ss;\ c\ \} \mapsto let}{\mathcal{V} \vdash \{\ x = a; ss;\ c\ \} \mapsto let} \qquad \textsc{dscf-this}\ \frac{}{\mathcal{V} \vdash this \mapsto this}$$

$$\textsc{dscf-if1}\ \frac{\begin{array}{c} x_i \in (\mathcal{A}[\![\{\ ss_1;\ a_1\ \}]\!] \cup \mathcal{A}[\![\{\ ss_2;\ a_2\ \}]\!]) \cap \mathcal{R}[\![\{\ ss_3;\ c_3\ \}]\!] \\ \mathcal{V}x_i = a'_i \qquad x \in \mathcal{R}[\![\{\ ss_3;\ c_3\ \}]\!] \qquad \mathcal{V}, x \leftarrow p, x_i \leftarrow p_i \vdash \{\ ss_3;\ c_3\ \} \mapsto let_3 \\ \mathcal{V} \vdash \{\ ss_1;\ k_3(a_1, x_i)\ \} \mapsto let_1 \qquad \mathcal{V} \vdash \{\ ss_2;\ k_3(a_2, x_i)\ \} \mapsto let_2 \end{array}}{\begin{array}{l} \mathcal{V} \vdash \{\ \mathtt{val}\ x = \mathtt{if}\ (a)\ \{\ ss_1;\ a_1\ \}\ \mathtt{else}\ \{\ ss_2;\ a_2\ \}; ss_3;\ c_3\ \} \\ \quad \mapsto \{\ \mathtt{def}\ k_1() = let_1; \mathtt{def}\ k_2() = let_2; \mathtt{def}\ k_3(p, p_i) = let_3;\ \mathtt{if}\ (a)\ k_1()\ \mathtt{else}\ k_2()\ \} \end{array}}$$

$$\textsc{dscf-if2}\ \frac{\begin{array}{c} x_i \in (\mathcal{A}[\![\{\ ss_1;\ a_1\ \}]\!] \cup \mathcal{A}[\![\{\ ss_2;\ a_2\ \}]\!]) \cap \mathcal{R}[\![\{\ ss_3;\ c_3\ \}]\!] \qquad \mathcal{V}x_i = a'_i \qquad x \notin \mathcal{R}[\![\{\ ss_3;\ c_3\ \}]\!] \\ \mathcal{V}, x_i \leftarrow p_i \vdash \{\ ss_3;\ c_3\ \} \mapsto let_3 \qquad \mathcal{V} \vdash \{\ ss_1;\ k_3(x_i)\ \} \mapsto let_1 \qquad \mathcal{V} \vdash \{\ ss_2;\ k_3(x_i)\ \} \mapsto let_2 \end{array}}{\begin{array}{l} \mathcal{V} \vdash \{\ \mathtt{val}\ x = \mathtt{if}\ (a)\ \{\ ss_1;\ a_1\ \}\ \mathtt{else}\ \{\ ss_2;\ a_2\ \}; ss_3;\ c_3\ \} \\ \quad \mapsto \{\ \mathtt{def}\ k_1() = let_1; \mathtt{def}\ k_2() = let_2; \mathtt{def}\ k_3(p_i) = let_3;\ \mathtt{if}\ (a)\ k_1()\ \mathtt{else}\ k_2()\ \} \end{array}}$$

$$\textsc{dscf-wdo}\ \frac{\begin{array}{c} x_i \in \mathcal{A}[\![\mathtt{while}\ (\{\ ss_1;\ a_1\ \})\ \{\ ss_2;\ a_2\ \}]\!] \\ \mathcal{V}x_i = a'_i \qquad \mathcal{V}, x_i \leftarrow p_i \vdash \{\ ss_3;\ c_3\ \} \mapsto let_3 \qquad \mathcal{V}, x_i \leftarrow p_i \vdash \{\ ss_2;\ k_1(x_i)\ \} \mapsto let_2 \\ \mathcal{V}, x_i \leftarrow p_i \vdash \{\ ss_1; \mathtt{def}\ k_2() = let_2; \mathtt{def}\ k_3() = let_3;\ \mathtt{if}\ (a_1)\ k_2()\ \mathtt{else}\ k_3()\ \} \mapsto let_1 \end{array}}{\mathcal{V} \vdash \{\ \mathtt{while}\ (\{\ ss_1;\ a_1\ \})\ \{\ ss_2;\ a_2\ \}; ss_3;\ c_3\ \} \mapsto \{\ \mathtt{def}\ k_1(p_i) = let_1;\ k_1(a'_i)\ \}}$$

$$\textsc{dscf-dow}\ \frac{\begin{array}{c} x_i \in \mathcal{A}[\![\mathtt{do}\ \{\ ss_2;\ a_2\ \}\ \mathtt{while}\ (\{\ ss_1;\ a_1\ \})]\!] \qquad \mathcal{V}x_i = a'_i \qquad \mathcal{V}, x_i \leftarrow p_i \vdash \{\ ss_3;\ c_3\ \} \mapsto let_3 \\ \mathcal{V}, x_i \leftarrow p_i \vdash \{\ ss_1; ss_2; \mathtt{def}\ k_3() = let_3;\ \mathtt{if}\ (a_2)\ k_1(x_i)\ \mathtt{else}\ k_3()\ \} \mapsto let_1 \end{array}}{\mathcal{V} \vdash \{\ \mathtt{do}\ \{\ ss_2;\ a_2\ \}\ \mathtt{while}\ (\{\ ss_1;\ a_1\ \}); ss_3;\ c_3\ \} \mapsto \{\ \mathtt{def}\ k_1(p_i) = let_1;\ k_1(a'_i)\ \}}$$

Fig. 18. Inference rules for the DSCF : $Source_{ANF} \Rightarrow Core_{ANF}$ transformation.

B CODE EXAMPLE

The following listing defines a module which implements a set of functions that compute confusion matrix values and an *F1*-score.

```
1  @lib
2  object eval {
3    type Hypothesis = DVector => Boolean
4    type TestBag[ID] = Bag[LDPoint[ID, Boolean]]
5
6    val TP = 0 // code for the true positives quadrant
7    val FP = 1 // code for the false positives quadrant
8    val FN = 2 // code for the false negatives quadrant
9    val TN = 3 // code for the true negatives quadrant
10
11   def apply[ID: Meta](h: Hypothesis)(xs: TestBag[ID]): Bag[LDPoint[ID, Int]] =
12     for (x <- xs) yield {
13       val quadrant = (if (h(x.pos)) 0 else 2) | (if (x.label) 0 else 1)
14       LDPoint(x.id, x.pos, quadrant)
15     }
16
17   def precision[ID: Meta](h: Hypothesis)(xs: TestBag[ID]): Double = {
18     val es = eval(h)(xs)
19     val tp = es.count(_.label == TP).toDouble
20     val fp = es.count(_.label == FP).toDouble
21     tp / (tp + fp)
22   }
23
24   def recall[ID: Meta](h: Hypothesis)(xs: TestBag[ID]): Double = {
25     val es = eval(h)(xs)
26     val tp = es.count(_.label == TP).toDouble
27     val fn = es.count(_.label == FN).toDouble
28     tp / (tp + fn)
29   }
30
31   def f1score[ID: Meta](h: Hypothesis)(xs: TestBag[ID]): Double = {
32     val p = precision(h)(xs)
33     val r = recall(h)(xs)
34     2.0 * (p * r) / (p + r)
35   }
36 }
```

Each function receives a `Hypothesis` function `h` which can classify a dense vector into a boolean space, and a `Bag` of labeled data points `xs` consisting of a generic `ID` and a `Boolean` label.

The `f1score` function is implemented in terms of the `precision` and `recall` functions (lines 32-33). These in turn are implemented in terms of the `apply` function (lines 18 and 25). The `apply` function maps over the points in `xs` and changes the label of each point `x` from the original `Boolean` value to a numeric value between `0` and `3` corresponding to the confusion matrix quadrant associated with `x` (lines 12-15). The `precision` and `recall` values compute different ratios between the value

counts in the different quadrants (lines 21 and 28). Finally, `f1score` computes the harmonic mean between the `precision` and `recall` values (line 34).

An simple quoted *Emma* term that uses the `eval` module might look as follows.

```
1 val f1score = onSpark {
2   val xs = Bag.readCSV[LDPoint[Int, Boolean]](INPUT_URL) // read a coll
3   val h = (pos: DVector) => pos(0) <= 0.5 // a very simple hypothesis function
4   eval.f1score(h)(xs)
5 }
6 print(s"The computed F1-score is ${f1score}")
```

Line 2 reads a bag of labeled data points with `Int` identifiers, and line 3 defines a simple hypothesis function which classifies data points based on the sign of the first component of their `pos` vector. Finally, line 4 calls the `f1score` method from the `eval` module. Since the `eval` module is marked with the `@lib` macro annotation, the `f1score` call in the quotation and the transitive calls of `precision`, `recall` and `apply` will be recursively inlined at the `f1score` call site in line 4. After this code expansion, the resulting Scala AST will be passed through the compiler pipeline defined by the `onSpark` macro. As part of this pipeline, the FOLD-FOREST-FUSION optimization will fuse all uses of the `xs Bag` into a single `fold` which simultaneously computes the `tp`, `fp`, and `fn` values defined in the `eval` listing at lines 19, 20, 26, and 27. The computed `f1score` will be returned as a result of the quoted expression and printed in the standard output (line 6).

C HOST LANGUAGE REQUIREMENTS

While the discussion in this paper is based on Scala, the presented EDSL design is not necessarily tied to the Scala ecosystem. In the following, we list some general requirements that need to be satisfied by every possible alternative host language H.

First, the targeted parallel dataflow APIs should be available as a library in H. This is required as the entire compilation pipeline is implemented as transformation of H terms. Consequently, we should be able to implement backend-specific implementations of the key *Emma* interfaces, such as `Bag`[A], `ComprehensionCombinators`, etc., in H.

Second, the concrete syntax of H should cover the following syntactic constructs: (a) value definitions and assignments, (b) method applications, (c) comprehensions, (d) `while` and `do` − `while` loops and `if` − `else` expressions, (e) lambda functions, and (f) recursive functions.

Depending on the set of constructs available in H, one might have to slightly modify the *Source* and *Core* language definitions. For example, if H does hot support type ascriptions, the $t : T$ syntactic form from Figure 3 and Figure 6 will not be needed. In such cases, one will also have to adapt the DSCF and the ANF transformations accordingly. In any case, the key idea to exclude host language recursive functions from *Source* and use them in order to model continuations in *Core* should be preserved irrespective of the choice of H.

Third, the language should provide facilities to (a) reify[15] designated source code terms as data structures of some H-specific AST type, and (b) compile and evaluate AST values. The *Emma* implementation should then provide bidirectional conversions between the syntactic forms available in the *Core* and *Source* languages and their corresponding AST encodings. Based on this functionality, one can then implement quotes such as the `onFlink` and `onSpark` macros presented in this paper.

Finally, H should be typed and its type system should provide support for parametric polymorphism. This is needed in order to be able to represent values of type `Bag`[A] in H. Further, the

[15]Reification is the process which converts an abstract idea (such as source code) into data (such as an AST).

type of each subterm should be available in the corresponding AST value upon reification. This is required since most of the *Emma* transformations specifically target terms of type Bag[A].

D NOTES ON THE COMPREHENSION NORMALIZATION RULE

The correctness of the UNNEST-HEAD from Figure 8 follows directly from the correctness of the $\mathcal{M}$-NORM-3 monad comprehension normalization rule proposed by Grust (cf. § 91 in [33]). For a comprehension where all generators bind values from the same monad type T (as is the case for *Emma* bag comprehensions where T = Bag), $\mathcal{M}$-NORM-3 has the following form.

$$\mathcal{M}\text{-NORM-3}\ \frac{}{\begin{array}{c}\texttt{for}\ \{\ qs_1\ ;\ x_3 \leftarrow \texttt{for}\ \{\ qs_2\ \}\ \texttt{yield}\ let_2\ ;\ qs_3\ \}\ \texttt{yield}\ let_1 \\ \mapsto \\ \texttt{for}\ \{\ qs_1\ ;\ qs_2\ ;\ [x_3 \coloneqq let_2]qs_3\ \}\ \texttt{yield}\ [x_3 \coloneqq let_2]let_1\end{array}}$$

In the following, we discuss how adapting $\mathcal{M}$-NORM-3 to the syntax of *Emma Core* results in the UNNEST-HEAD variant presented in Figure 8.

First, we need to capture the context in which the matched outer comprehension is defined.

$$\{\ vdefs_1;\ \texttt{val}\ x_1 = \texttt{for}\ \{\ qs_1\ ;\ x_3 \leftarrow \ldots\ ;\ qs_3\ \}\ \texttt{yield}\ let_1;\ vdefs_3;\ kdefs_1;\ c\ \}$$

Second, due to the constrained syntax of comprehension generators (cf. Figure 6), instead of

$$x_3 \leftarrow \texttt{for}\ \{\ qs_2\ \}\ \texttt{yield}\ let_2$$

we have to match expressions of the general form.

$$x_3 \leftarrow \{\ vdefs_2;\ kdefs_2;\ x_2\ \}$$

The rule triggers if x_2 is a value definition of a comprehension with the general form

$$\texttt{val}\ x_2 = x \leftarrow \texttt{for}\ \{\ qs_2\ \}\ \texttt{yield}\ let_2$$

and x_2 is referenced exactly once – in the x_3 generator.

To simplify the rule, we consider only cases where (*i*) $kdefs_2$ is empty (i.e., the generator right-hand side does not contain local control-flow structure), and (*ii*) the x_2 definition is part of either $vdefs_1$ or $vdefs_2$. In practice, however, these assumptions do not constitute a serious limitation, as code examples that violate them either cannot be normalized or represent extremely degenerate input programs.

Things are further complicated by the fact that in order to eliminate the x_3 generator, in addition to the x_2 comprehension we now also have to unnest $vdefs_2$. To do that, we partition $vdefs_2$ in two subsets – values that do and do not depend on generator symbols defined in qs_1, denoted respectively as $vdefs_2^D$ and $vdefs_2^I$. Definitions contained in $vdefs_2^I$ are prepended just before the x_1 definition in the enclosing *let* block. Definitions in $vdefs_2^D$ need to be replicated in *let*-blocks contained in the qualifier sequence qs_3 and in the head of the outer comprehension let_1.

Finally, in order to maintain the $Core_{ANF}$ form, term substitution is more involved. Instead of $[x_3 \coloneqq let_2]\cdot$, we use the auxiliary $fix(\cdot)$ function which transforms its input *let* block as described in Section 6.3. We end up with the UNNEST-HEAD rule from Section 6.3.

$$\textsc{UnnestHead}\ \frac{\begin{array}{c}x_1 : \mathrm{M}A \qquad \texttt{val}\ x_2 = \texttt{for}\ \{\ qs_2\ \}\ \texttt{yield}\ let_2 \in vdefs_1 +\!\!+ vdefs_2 \qquad uses(x_2) = 1 \\ (vdefs_2^I, vdefs_2^D) \coloneqq split(remove(x_2, vdefs_2), qs_1) \\ qs' \coloneqq qs_1 +\!\!+ qs_2 +\!\!+ qs_3.map(fix) \qquad let_1' \coloneqq fix(let_1) \qquad vdefs_1' \coloneqq remove(x_2, vdefs_1)\end{array}}{\begin{array}{c}\{\ vdefs_1; \texttt{val}\ x_1 = \texttt{for}\ \{\ qs_1; x_3 \leftarrow \{\ vdefs_2;\ x_2\ \}; qs_3\ \}\ \texttt{yield}\ let_1; vdefs_3;\ kdefs;\ c\ \} \\ \mapsto \{\ vdefs_1'; vdefs_2^I; \texttt{val}\ x_1 = \texttt{for}\ \{\ qs'\ \}\ \texttt{yield}\ let_1'; vdefs_3;\ kdefs;\ c\ \}\end{array}}$$

E NOTES ON THE COMPREHENSION COMBINATION RULES

First, note that the Bag monad is commutative in the sense defined by Wadler [66]. More precisely, this means that

$$\texttt{for}\ \{\ q_1\ ;\ q_2\ \}\ \texttt{yield}\ let \quad = \quad \texttt{for}\ \{\ q_2\ ;\ q_1\ \}\ \texttt{yield}\ let$$

for any head expression *let* and any pair of qualifiers q_1 and q_2 such that q_2 binds no free variables of q_1 and vice versa.

Taking this into account, the correctness of COM-FILTER can be shown as follows. The matched comprehension has the general form

$$\texttt{for}\ \{\ qs_1;\ x \leftarrow xs;\ qs_2;\ \texttt{if}\ p;\ qs_3\ \}\ \texttt{yield}\ let$$

Because the premise of the rule assumes that $\mathcal{R}[\![p]\!] \cap \mathcal{G}[\![qs_1 +\!\!+ qs_2]\!] = \emptyset$, we can safely pull the `if` p guard before the qs_2 generator sequence.

$$\texttt{for}\ \{\ qs_1;\ x \leftarrow xs;\ \texttt{if}\ p;\ qs_2;\ qs_3\ \}\ \texttt{yield}\ let$$

We now apply the $\mathcal{M}$-NORM-3 rule from the previous section in the opposite direction, pulling both the generator $x \leftarrow xs$ and the subsequent guard `if` p into a nested comprehension with a trivial head expression { x }.

$$\texttt{for}\ \{\ qs_1;\ x \leftarrow \texttt{for}\ \{\ x \leftarrow xs;\ \texttt{if}\ p\ \}\ \texttt{yield}\ \{\ x\ \};\ qs_2;\ qs_3\ \}\ \texttt{yield}\ let$$

Finally, we apply the RES-FILTER rule from Figure 7 in the reverse direction, desugaring the nested comprehension to a `withFilter` call.

$$\texttt{for}\ \{\ qs_1;\ x \leftarrow xs.\texttt{withFilter}(x \Rightarrow p);\ qs_2;\ qs_3\ \}\ \texttt{yield}\ let$$

The correctness of the remaining comprehension combination rules can be established by similar arguments.

LIST OF ACRONYMS

ADT **A**lgebraic **D**ata **T**ype.
ANF **A**dministrative **N**ormal **F**orm.
API **A**pplication **P**rogramming **I**nterface.
AST **A**bstract **S**yntax **T**ree.

CCI **c**ache-call **i**nsertion.
CL **C**omprehension **L**anguage.

DBMS **D**atabase **M**anagement **S**ystem.
DIQL **D**ata **I**ntensive **Q**uery **L**anguage.
DSH **D**atabase-Supported **H**askell.
DSL **D**omain **S**pecific **L**anguage.

EDSL **E**mbedded **D**omain **S**pecific **L**anguage.

FGF **f**old-group **f**usion.

GPL **G**eneral-purpose **P**rogramming **L**anguage.

IDE **I**ntegrated **D**evelopment **E**nvironment.
IMDb **I**nternet **M**ovie **D**atabase.
IR **I**ntermediate **R**epresentation.

JIT **j**ust-in-time.
JVM **J**ava **V**irtual **M**achine.

LINQ **L**anguage-Integrated **Q**uery.
LMS **L**ightweight **M**odular **S**taging.

ML **M**achine **L**earning.

NIS **n**ative **i**terations **s**pecialization.

OQL **O**bject **Q**uery **L**anguage.

RAS **r**elational **a**lgebra **s**pecialization.
RDBMS **R**elational **D**atabase **M**anagement **S**ystem.
RDF **R**adial **D**istribution **F**unction.

SQL **S**tructured **Q**uery **L**anguage.
SSA **S**tatic **S**ingle **A**ssignment.
UDF **U**ser-Defined **F**unction.
UDT **U**ser-Defined **T**ype.

REFERENCES

[1] Stefan Ackermann, Vojin Jovanovic, Tiark Rompf, and Martin Odersky. 2012. Jet: An embedded DSL for high performance big data processing. In *International Workshop on End-to-end Management of Big Data (BigData 2012)*.

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *PVLDB* 8, 12 (2015), 1792–1803.

[3] Alexander Alexandrov, Asterios Katsifodimos, Georgi Krastev, and Volker Markl. 2016. Implicit Parallelism through Deep Language Embedding. *SIGMOD Record* 45, 1 (2016), 51–58.

[4] Alexander Alexandrov, Andreas Kunft, Asterios Katsifodimos, Felix Schüler, Lauritz Thamsen, Odej Kao, Tobias Herb, and Volker Markl. 2015. Implicit Parallelism through Deep Language Embedding. In *SIGMOD Conference*. 47–61.

[5] Andrew W. Appel. 1998. SSA is Functional Programming. *SIGPLAN Notices* 33, 4 (1998), 17–20.

[6] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). 1383–1394.

[7] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. 2010. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum (Eds.). 119–130.

[8] Kevin Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh Carl-Christian Kanne, Fatma Ozcan, and Eugene J. Shekita. 2011. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB* (2011).

[9] Richard S. Bird and Oege de Moor. 1997. *Algebra of programming*.

[10] Richard S et al. Bird. 1987. An introduction to the theory of lists. *Logic of programming and calculi of discrete design* 36 (1987), 5–42.

[11] P. Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. 2014. Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. *PVLDB* 7, 13 (2014), 1441–1451.

[12] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. 1994. Comprehension Syntax. *SIGMOD Record* (1994).

[13] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. 1995. Principles of Programming with Complex Objects and Collection Types. *Theor. Comput. Sci.* 149, 1 (1995), 3–48.

[14] Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A Structured English Query Language. In *Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, May 1-3, 1974, 2 Volumes*, Randall Rustin (Ed.). 249–264.

[15] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. 2010. FlumeJava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). 363–375.

[16] James Cheney, Sam Lindley, and Philip Wadler. 2013. A practical theory of language-integrated query. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). 403–416.

[17] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970), 377–387.

[18] Phil Trinder David A. Watt. 1991. Towards a Theory of Bulk Types. (1991).

[19] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*. 137–150.

[20] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. 2012. Spinning Fast Iterative Data Flows. *PVLDB* 5, 11 (2012), 1268–1279.

[21] Leonidas Fegaras. 1994. *A uniform calculus for collection types.* Technical Report. Oregon Graduate Institute. 94–030 pages.

[22] Leonidas Fegaras and Ashiq Imran. 2017. Compile-Time Code Generation for Embedded Data-Intensive Query Languages. (July 2017). Under submission.

[23] Leonidas Fegaras, Chengkai Li, and Upa Gupta. 2012. An optimization framework for map-reduce queries. In *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*,

Elke A. Rundensteiner, Volker Markl, Ioana Manolescu, Sihem Amer-Yahia, Felix Naumann, and Ismail Ari (Eds.). 26–37.

[24] Leonidas Fegaras and David Maier. 1995. Towards an Effective Calculus for Object Query Languages. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.*, Michael J. Carey and Donovan A. Schneider (Eds.). 47–58.

[25] E. Forgy. 1965. Cluster Analysis of Multivariate Data: Efficiency versus Interpretability of Classification. *Biometrics* 21, 3 (1965), 768–769.

[26] Johann Christoph Freytag. 1987. A Rule-Based View of Query Optimization. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, Umeshwar Dayal and Irving L. Traiger (Eds.). 173–180.

[27] Peter Gassner, Guy M. Lohman, K. Bernhard Schiefer, and Yun Wang. 1993. Query Optimization in the IBM DB2 Family. *IEEE Data Eng. Bull.* 16, 4 (1993), 4–18.

[28] Luca M Ghiringhelli, Christian Carbogno, Sergey Levchenko, Fawzi Mohamed, Georg Huhs, Martin Lüders, Micael Oliveira, and Matthias Scheffler. 2016. Towards a Common Format for Computational Material Science Data. *arXiv preprint arXiv:1607.04738* (2016).

[29] Jeremy Gibbons. 2016. Comprehending Ringads - For Phil Wadler, on the Occasion of his 60th Birthday. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.), Vol. 9600. 132–151.

[30] Jeremy Gibbons and Nicolas Wu. 2014. Folding domain-specific languages: deep and shallow embeddings (functional Pearl). In *ICFP*. 339–347.

[31] Andrew John Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*, John Williams (Ed.). ACM, 223–232. DOI: https://doi.org/10.1145/165180.165214

[32] George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. 2010. Haskell Boards the Ferry - Database-Supported Program Execution for Haskell. In *Implementation and Application of Functional Languages - 22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers (Lecture Notes in Computer Science)*, Jurriaan Hage and Marco T. Morazán (Eds.), Vol. 6647. 1–18.

[33] Torsten Grust. 1999. *Comprehending Queries.* Ph.D. Dissertation. Universität Konstanz.

[34] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. 2009. FERRY: database-supported program execution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). 1063–1066.

[35] Torsten Grust, Jan Rittinger, and Tom Schreiber. 2010. Avalanche-Safe LINQ Compilation. *PVLDB* 3, 1 (2010), 162–172.

[36] Torsten Grust and Marc H. Scholl. 1999. How to Comprehend Queries Functionally. *J. Intell. Inf. Syst.* 12, 2-3 (1999), 191–218.

[37] Zhenyu Guo, Xuepeng Fan, Rishan Chen, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Chang Liu, Wei Lin, Jingren Zhou, and Lidong Zhou. 2012. Spotting Code Optimizations in Data-Parallel Pipelines through PeriSCOPE. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). 121–133.

[38] Paul Hudak. 1996. Building Domain-Specific Embedded Languages. *ACM Comput. Surv.* 28, 4es (1996), 196. DOI: https://doi.org/10.1145/242224.242477

[39] Fabian Hueske, Mathias Peters, Matthias Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. 2012. Opening the Black Boxes in Data Flow Optimization. *PVLDB* 5, 11 (2012), 1256–1267.

[40] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*, Paulo Ferreira, Thomas R. Gross, and Luís Veiga (Eds.). 59–72.

[41] Simon L. Peyton Jones and Philip Wadler. 2007. Comprehensive comprehensions. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, Gabriele Keller (Ed.). 61–72.

[42] Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. 2014. Yin-yang: concealing the deep embedding of DSLs. In *GPCE*. 73–82.

[43] Guy L. Steele Jr. 2009. Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, Graham Hutton and Andrew P. Tolmach (Eds.). 1–2.

[44] Aljoscha Krettek. 2015. *Using Meta-Programming to Analyze and Rewrite Domain-Specific Program Code.* Master's thesis. TU Berlin.

[45] Joachim Lambek. 1993. Least Fixpoints of Endofunctors of Cartesian Closed Categories. *Mathematical Structures in Computer Science* 3, 2 (1993), 229–257.

[46] Roland Leißa, Klaas Boesche, Sebastian Hack, Richard Membarth, and Philipp Slusallek. 2015. Shallow embedding of DSLs via online partial evaluation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2015, Pittsburgh, PA, USA, October 26-27, 2015*, Christian Kästner and Aniruddha S. Gokhale (Eds.). ACM, 11–20. DOI: https://doi.org/10.1145/2814204.2814208

[47] Johannes Luong, Dirk Habich, and Wolfgang Lehner. 2017. AL: unified analytics in domain specific terms. In *Proceedings of The 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, September 1, 2017*, Tiark Rompf and Alexander Alexandrov (Eds.). 7:1–7:9.

[48] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). 135–146.

[49] Erik Meijer, Brian Beckman, and Gavin M. Bierman. 2006. LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis (Eds.). 706.

[50] Guido Moerkotte and Thomas Neumann. 2006. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). 930–941.

[51] Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92.

[52] Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything old is new again: quoted domain-specific languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Martin Erwig and Tiark Rompf (Eds.). ACM, 25–36. DOI: https://doi.org/10.1145/2847538.2847541

[53] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*. 1099–1110.

[54] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*, Eelco Visser and Jaakko Järvi (Eds.). 127–136.

[55] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1.*, Philip A. Bernstein (Ed.). 23–34.

[56] David B Skillicorn. 1993. The Bird-Meertens formalism as a parallel model. In *Software for Parallel Computation*. 120–133.

[57] D. B. Skillicorn. 1993. Structuring data parallelism using categorical data types. In *Proc. Workshop Programming Models for Massively Parallel Computers*. 110–115.

[58] Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embedded Comput. Syst.* 13, 4s (2014), 134:1–134:25.

[59] Akihiko Takano and Erik Meijer. 1995. Shortcut Deforestation in Calculational Form. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, John Williams (Ed.). ACM, 306–313. DOI: https://doi.org/10.1145/224164.224221

[60] Val Tannen, Peter Buneman, and Shamim A. Naqvi. 1991. Structural Recursion as a Query Language. In *Database Programming Languages: Bulk Types and Persistent Data. 3rd International Workshop, August 27-30, 1991, Nafplion, Greece, Proceedings*, Paris C. Kanellakis and Joachim W. Schmidt (Eds.). 9–19.

[61] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB* 2, 2 (2009), 1626–1629.

[62] Phil Trinder and Philip Wadler. 1989. Improving list comprehension database queries. In *TENCON'89. Fourth IEEE Region 10 International Conference*. IEEE, 186–192.

[63] Philip W. Trinder. 1991. Comprehensions, a Query Notation for DBPLs. In *Database Programming Languages: Bulk Types and Persistent Data. 3rd International Workshop, August 27-30, 1991, Nafplion, Greece, Proceedings*, Paris C. Kanellakis and Joachim W. Schmidt (Eds.). 55–68.

[64] Alexander Ulrich and Torsten Grust. 2015. The Flatter, the Better: Query Compilation Based on the Flattening Transformation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). 1421–1426.
[65] Philip Wadler. 1990. Notes on Monads and Ringads. (September 1990). Internal document, Computing Science Dept. Glasgow University.
[66] Philip Wadler. 1992. Comprehending Monads. *Mathematical Structures in Computer Science* (1992).
[67] Philip Wadler. 1995. How to Declare an Imperative. In *Logic Programming, Proceedings of the 1995 International Symposium, Portland, Oregon, USA, December 4-7, 1995*, John W. Lloyd (Ed.). 18–32.
[68] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *HotCloud*.
[69] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Åke Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: parallel databases meet MapReduce. *VLDB J.* 21, 5 (2012), 611–636.

Received March 2018; revised September 2018