Representations and Optimizations for Embedded Parallel Dataflow Languages

vorgelegt von
M.Sc.
Alexander Alexandrov
geb. in Sofia, Bulgarien

von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Odej Kao
Gutachter:	Prof. Dr. Volker Markl
Gutachterin:	Prof. Dr. Mira Mezini
Gutachter:	Prof. Dr. Torsten Grust

Tag der wissenschaftlichen Aussprache: 31. Oktober 2018

Berlin 2019

In memory of my grandfather,
who taught me how to count when I was very young
and Prof. Hartmut Ehrig,
who taught me how to comprehend counting twenty years later.

Acknowledgments

I would like to express my gratitude to the people who made this dissertation possible.

First and foremost, I would like to thank my advisor Prof. Dr. Volker Markl. He offered me the chance to work in the area of data management and encouraged me to search for a motivating topic. Throughout my time at the Database and Information Systems Group at TU Berlin, his constant engagement and valuable advice allowed me to substantially improve the quality of my research.

I am also deeply indebted to everybody who contributed to the *Emma* project. Asterios Katsifodimos and Andreas Kunft showed tremendous dedication and work ethic and played an essential role in bringing the original SIGMOD 2015 submission to an acceptable shape under a very tight deadline. Georgi Krastev was instrumental in shaping the design and implementation of the compiler internals. Without his passion for functional programming and sharp eye for elegant API design, the software artifact accompanying this thesis would undoubtedly have ended up in a much more rudimentary form. Gábor Gévay influenced the story presented in this thesis with a number of incisive comments. Most notably, he rigorously pointed out that state-of-the-art solutions fall into the category of deeply embedded DSLs, which forced me to pinpoint quotation-based embedding as the crux to the proposed solution. Andreas Salzmann developed the GUI for the demonstrator, and Felix Schüler and Bernd Louis contributed a number of algorithms to the *Emma* library.

This work represents a natural fusion between two distinct lines of research. Stephan Ewen and Fabian Hüske developed the original PACT programming model, and I was lucky enough to work with both of them during my time at DIMA. The adopted categorical approach highlights the timeless relevance of the foundational research conducted by Phil Wadler, Peter Buneman and Val Tannen, and Torsten Grust in the 1990s. The intimate connection between the two areas was pointed out by Alin Deutsch during a visit at UC San Diego in the autumn of 2012.

Last but not least, I am also grateful to my family and friends for their continuous love and support, to all past and future teachers of mine for their shared knowledge, and to Katya Tasheva, Emma Greenfield, and Petra Nachtmanova for the music.

Declaration of Authorship

I, Alexander Alexandrov, declare that this thesis, titled "Representations and Optimizations for Embedded Parallel Dataflow Languages", and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Berlin, February 12, 2019 .....

Abstract

Parallel dataflow engines such as Apache Hadoop, Apache Spark, and Apache Flink have emerged as an alternative to relational databases more suitable for the needs of modern data analysis applications. One of the main characteristics of these systems is their scalable programming model, based on distributed collections and parallel transformations. Notable examples are Flink's `DataSet` and Spark's `RDD` programming abstractions.

The programming model is typically realized as an eDSL – a domain specific language embedded in a general-purpose host language such as Java, Scala, or Python. This approach has several advantages over traditional stand-alone DSLs such as SQL or XQuery. First, it allows for reuse of linguistic constructs from the host language – for example, anonymous functions syntax, value definitions, or fluent syntax via method chaining. This eases the learning curve for developers already familiar with the host language syntax. 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 development effort for dataflows that go beyond pure SQL and require domain-specific logic, for example for text or image pre-processing.

At the same time, state-of-the-art parallel dataflow eDSLs exhibit a number of shortcomings. First, one of the main advantages of a stand-alone DSL such as SQL – the high-level, declarative `Select-From-Where` syntax – is either lost or mimicked in a non-standard way. Second, execution aspects such as caching, join order, and partial aggregation need to be decided by the programmer. Automatic optimization is not possible due to the limited program context reflected in the eDSL intermediate representation (IR).

In this thesis, we argue that these limitations are a side effect of the adopted type-based embedding approach. As a solution, we propose an alternative eDSL design based on quasi-quotations. We present a DSL embedded in Scala and discuss its compiler pipeline, IR, 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 `for`-comprehensions can be used to encode `Select-From-Where` expressions in a standard way. At the IR level, maintaining comprehensions as a first-class citizen can be used to simplify the analysis 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 such as SQL.

Zusammenfassung

Parallele Datenflusssysteme wie Apache Hadoop, Apache Spark und Apache Flink haben sich als Alternative von relationalen Datenbanken etabliert, die für die Anforderungen moderner Datenanalyseanwendungen besser geeignet ist. Zu den Hauptmerkmalen dieser Systeme gehört ein auf verteilten Datenkollektionen und parallelen Transformationen basierendes Programmiermodell. Beispiele dafür sind die `DataSet` und `RDD` Programmierschnittstellen von Flink und Spark.

Diese Schnittstellen werden in der Regel als eDSLs realisiert, d.h. als domänenspezifische Sprachen, die in einer Hostsprache wie Java, Scala oder Python eingebettet sind. Dieser Ansatz bietet mehrere Vorteile gegenüber herkömmlichen externen DSLs wie SQL oder XQuery. Zum einen kann man bei einer eDSL syntaktische Konstrukte aus der Host-Sprache wiederverwenden. Dies verringert die Lernkurve für Entwickler, die bereits mit der Syntax der Hostsprache vertraut sind. Zum anderen ermöglicht der Ansatz eine nahtlose Integration von Bibliotheksmethoden, die in der Hostsprache verfügbar sind, und reduziert somit den Entwicklungsaufwand für Datenflüsse, die über reines SQL hinausgehen und domänenspezifische Logik erfordern.

Gleichzeitig weisen eDSLs wie `DataSet` und `RDD` eine Reihe von Nachteilen auf. Erstens ist einer der Hauptvorteile von externen DSLs wie SQL - die deklarative `Select-From-Where`-Syntax - entweder verloren oder auf eine nicht-standardisierte Weise nachgeahmt. Zweitens werden Ausführungsaspekte wie Caching, Join-Reihenfolge und verteilte Aggregate vom Programmierer manuell festgelegt. Eine automatische Optimierung ist aufgrund des begrenzten Programmkontexts in der eDSL-Zwischenrepräsentation nicht möglich.

Wir zeigen, dass diese Einschränkungen als Nebeneffekt des auf Typen basierenden Einbettungsansatzes verursacht werden. Als Lösung schlagen wir ein alternatives Design vor, das auf Quasi-Quotations basiert. Wir präsentieren eine Scala eDSL und diskutieren deren Compiler, Zwischenrepräsentation, sowie einigen von den ermöglichten Optimierungen. Als Grundlage für das verteilte Datenmodell benutzen wir den algebraischen Typ von Kollektionen in Union-Repräsentation, und für die parallele Datenverarbeitung – die damit verbundenen strukturelle Rekursion und Monade. Auf der Quellcode-Ebene kann man Comprehensions über die Monade verwenden, um `Select-From-Where` Ausdrücke in einer Standardform zu kodieren. In der Zwischenrepräsentation bieten Comprehensions eine Basis, auf der man Datenflussoptimierungen einfacher gestalten kann. Das vorgeschlagene Design vereinigt somit die Vorteile von eingebetteten parallelen Datenfluss-DSLs mit der deklarativen Natur und Optimierungspotenzial von externen DSLs wie SQL.

Contents

List of Figures

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 in [Cod70]. The notion of *processing* has been traditionally associated with relational engines – specialized runtimes for efficient evaluation of relational algebra programs. Finally, 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 [CB74] used in the Structured Query Language (SQL), and (*ii*) with data-driven query compilation techniques [SAC+79]. Data management solutions based the *declarative data processing* paradigm therefore traditionally interface with their clients through a stand-alone Domain Specific Language (DSL), most commonly SQL.

While SQL is easy to teach and straight-forward to use for simple descriptive analytics, it is not so well-suited for more advanced pipelines. The limitations of SQL manifest themselves most notably in domains like data integration or predictive data analysis. Programs in these domains are characterized by dataflow features not directly supported by SQL, such as dataflows with iterative or nested structure or application-specific element-wise transformations. To illustrate this, imagine a text processing pipeline which clusters text documents using an algorithm such as *k-means*. Conceptually, the input of such pipeline is a collection (document corpus) of nested collections (the words in a specific document). The first part of the pipeline therefore has to operate on this nested collection 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 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 (usually offered as non-standard extensions) such as User-Defined Types (UDTs), User-Defined Functions (UDFs), and control-flow primitives such as the ones provided by PL/SQL. Language-integrated SQL technologies such as Microsoft's Language-Integrated Query (LINQ) mitigate some of these issues, but do not deal well with the iterative dataflows characteristic for most data analysis pipelines.

In contrast, systems such as Apache Hadoop, Apache Spark, and Apache Flink offer more flexibility for programming 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 guarantee certain degrees of data-parallelism that are 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 available in the host language.

Despite this advantage, state-of-the-art Embedded Domain Specific Languages (eDSLs) offered by Spark and Flink also exhibit some common problems. First, one of the main benefits of a Stand-alone Domain Specific Language (sDSL) 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-standard way. Second, execution aspects such as caching, join order, and partial aggregation need to be controlled and 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, the construction of efficient pipelines requires programmers with deep understanding of the underlying dataflow engine. Further, mixing in physical execution aspects in the application code increases its long-term maintenance cost.

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

(**C1**) We analyze state-of-the-art eDSLs for parallel collection processing and identify type-based embedding as the root cause for a set of commonly exhibited deficiencies.

(**C2**) 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.

(C3) As a solution to the problems highlighted in (**C1**), we propose a Scala DSL for parallel collection processing based on *quasi-quotations*[1]. We discuss the eDSL concrete syntax and Application Programming Interface (API), its abstract syntax IR, and a compiler frontend that mediates between the two.

(C4) Building on top of the IR from (**C3**), we develop optimizations that cannot be attained by parallel dataflow eDSLs with type-based embedding (e.g., fold-based fusion, operator specialization, and auto-caching) and highlight their relation to traditional optimizations from the compiler or data management domains.

(C5) We implement backends that offload data-parallel computation on Apache Spark and Apache Flink, and demonstrate that the performance of code produced by these backends is on par with that of hand-optimized Spark and Flink dataflows. The automatic optimizations from (**C4**) thereby lower the requirements on the programmer. At the same time, separating the user-facing source language, the IR, and the backend parallel dataflow co-processor ensures performance portability.

(C6) We argue about the utility of monad comprehensions as first-class citizen. 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., with `for`-comprehensions in Scala. At the IR level, treating comprehensions as a primitive building block simplifies the definition and analysis of holistic dataflow optimizations in the presence of nesting and control flow.

The proposed design can therefore be seen as a step towards reconciling the flexibility of modern eDSLs for parallel collection processing with the declarative nature and optimization potential traditionally associated with sDSLs such as xQuery and SQL.

The thesis is structured as follows. Chapter 2 reviews state-of-the-art technology and the research problem, while Chapter 3 outlines the proposed solution. Chapter 4 provides the background necessary for the methodology we employ towards our solution. Chapter 5 presents the abstract syntax and core API of *Emma* – a quotation-based DSL for parallel collection processing embedded in Scala. Chapter 6 presents *Emma Core* – an IR suitable for optimization, and a transformation from *Emma Source* to *Emma Core*. Chapter 7 develops optimizations on top of *Emma Core*. Chapter 8 discusses possible implementation infrastructures and some aspects of our prototype implementation.

[1] In quotation-based DSLs, terms are delimited not by their type, but by an enclosing function which can access and transform the Abstract Syntax Tree of its arguments. For example, in the Scala expression

`onSpark { ... code ... }`

the `onSpark` quasi-quote delimits a Scala `code` snippet that will be automatically optimized and evaluated on Spark by the eDSL compiler presented in this thesis.

Chapter 9 highlights the impact and importance of the proposed optimizations through an experimental evaluation. Chapter 10 reviews related work. Finally, Chapter 11 concludes and discusses future research directions.

The material presented in this thesis is based on the following publications. The `Bag` API from Section 5.3, the comprehension compilation scheme from Section 7.1.2, and the FOLD-GROUP-FUSION optimizing transformation from Section 7.2.2 were first published at the SIGMOD 2015 conference [AKK+15]. A revised version of this work appeared in the SIGMOD Record journal in 2016 [AKKM16], and a demonstrator was showcased at the SIGMOD 2016 and BTW 2017 conferences [AKL+17, ASK+16]. Notably, the publications listed above no not rely on the *Emma Core* IR discussed in Chapter 6. Instead, the suggested implementation methodology is based on vanilla Scala Abstract Syntax Trees (ASTs) and an auxiliary "comprehension layer" developed on top of the Scala AST representation. Using *Emma Core* as a basis for the optimizations discussed in Chapter 7 represents a more general approach, as it decouples the eDSL IR from the IR of host language IR. For example, the FOLD-GROUP-FUSION optimization discussed in [AKK+15, AKKM16] is presented only in conjunction with the BANANA-SPLIT law. The variant presented here, on the other hand, combines the BANANA-SPLIT and the CATA-MAP-FUSION laws as a dedicated, FOLD-FOREST-FUSION transformation based on the *Emma Core* IR. The metaprogramming API discussed in Section 8.4 was developed jointly with Georgi Krastev in 2016-2017 and is has not been published before. A shortened version of the material presented in this thesis (excluding Section 4.1 and Chapter 8) has been reworked as a journal paper and is currently under submission.

2 State of the Art and Problems

In this section we review open problems with state-of-the-art technology. We begin by introducing common notions related to the implementation (Section 2.1) and design (Section 2.2) of DSLs which are relevant for the subsequent discussion. We then provide a historical perspective of the evolution of embedded DSLs for scalable data analysis (Section 2.3), highlighting the benefits and problems of the various implementation approaches by example.

2.1 DSL Implementation Approaches

The DSL classes discussed below are depicted on Figure 2.1, with definitions adapted from [GW14]. With regard to their implementation approach and relation to General-purpose Programming Languages (GPLs), DSLs can be divided in two classes – *stand-alone* and *embedded*.

Stand-alone Domain Specific Languages (sDSLs) define their own syntax and semantics. The main 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, a stand-alone DSL requires (*i*) building a dedicated parser, type-checker, and a compiler or interpreter, (*ii*) additional tools for IDE integration, debugging, and documentation, and (*iii*) off-the-shelf functionality in the form of a standard or third-party libraries. Examples of widely adopted stand-alone DSLs are Verilog and SQL.

Embedded Domain Specific Languages (eDSLs) are embedded into a GPL usually referred to as *host language*. This approach can be seen as more pragmatic compared do sDSLs for at least two reasons. First, it allows to reuse the concrete syntax of the host language. Second, it also allows to reuse existing host-language infrastructure such as Integrated Development Environments (IDEs), debugging tools, and dependency management.

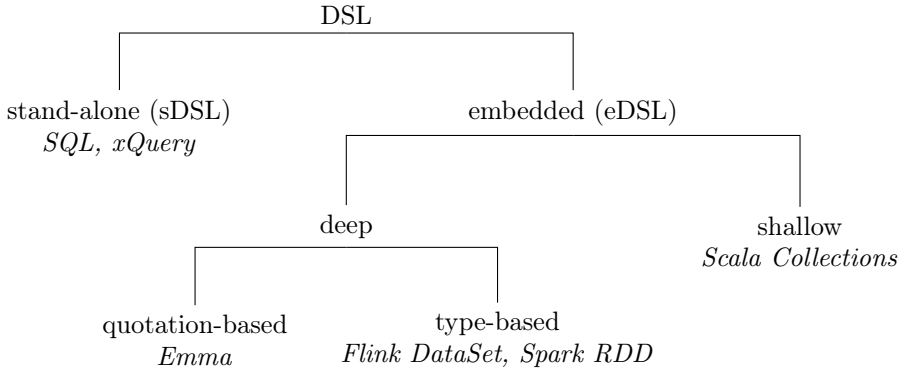


Figure 2.1: Classification of DSLs. Examples in each class are given in *italic*.

Based on the embedding strategy, eDSLs can be further differentiated into two sub-classes. With *shallow embedding*, DSL terms are implemented *directly* by defining their semantics in host language constructs. With *deep embedding*, DSL terms are implemented *reflectively* by constructing an IR of themselves. The IR is subsequently optimized and either interpreted or compiled.

Finally, the method used for IR construction in deeply embedded DSLs yields two more sub-classes. With the *type-based* approach, the eDSL consists of dedicated types, and operations on these types incrementally construct its IR. Host language terms that belong to the eDSL are thereby delimited by their type. With the *quotation-based* approach, the eDSL derives its IR from a host-language AST using some form of metaprogramming facilities offered by the host language. Host language terms that belong to the eDSL are thereby delimited by the surrounding quasi-quotes.

2.2 eDSL Design Objectives

Sharing its concrete syntax with host language is an important property of eDSLs that can be utilized to improve their learning curve and adoption. In that regard, eDSL design should be guided by two principles. The first is to *maximize linguistic reuse* – that is, to exploit the programmer's familiarity with syntactic conventions and tools of the host language in carrying those over to the eDSL as much as possible. The second is to *minimize linguistic noise* – that is, to reduce the amount of idiosyncratic constructs specific to the eDSL as much as possible. At the same time, the eDSL should aim to improve developer productivity and at the same time maximize the runtime performance through advanced, domain-specific optimizations.

2.3 Parallel Dataflow DSLs – Evolution and Problems

Based on the discussion above, we can review the evolution of parallel dataflow DSLs, outline limitations of state-of-the-art solutions, and discuss current solution strategies.

2.3.1 Origins: MapReduce & Pregel

In its early days, Google faced problems with two integral parts of its data engineering pipeline – (*i*) computing an inverted index and (*ii*) ranking the pages in a crawled Web corpus. Conceptually, the input for the first task is a collection of documents identified by an URL, and the goal is to tokenize the text content of each document into distinct words, pairing each word with the URLs of the documents where this word occurs. The input and output can therefore be seen as collections with the following element types[1].

$$\begin{aligned} \texttt{Document} &= (\texttt{url} : \texttt{URL}) \times (\texttt{content} : \texttt{String}) && \text{input element type} \\ \texttt{Index} &= (\texttt{word} : \texttt{String}) \times (\texttt{urls} : \texttt{URL}^*) && \text{output element type} \end{aligned}$$

The input for the second task is a collection of URLs with adjacent URLs induced by their outbound hyperlinks, and the output – an assignment of ranks to each URL – is computed by iteratively re-distributing the current rank across adjacent links until convergence. The input and output element types look as follows.

$$\begin{aligned} \texttt{Page} &= (\texttt{url} : \texttt{URL}) \times (\texttt{links} : \texttt{URL}^*) && \text{input element type} \\ \texttt{Rank} &= (\texttt{url} : \texttt{URL}) \times (\texttt{rank} : \texttt{Double}) && \text{output element type} \end{aligned}$$

Initially, Google attempted to implement both tasks in a relational database. This approach, however, had two major problems. First, handling the extreme input size required a distributed setup with thousands of nodes, which significantly increased the risk of a node failure during job execution. However, distributed database technology was not designed to resiliently execute long-running queries at such scale in the presence of frequent failures, and the pipeline was breaking too often. Second, the relational data model and query language were not the best fit for the tasks at hand. Due to their nested structure, neither `Index` nor `Rank` could be encoded natively as SQL tables. Additionally, expressing the two tasks as queries required non-standard SQL extensions, e.g., a `tokenize` UDF and an `unnest` operator for the inverted index task, and support for iterative dataflows for the page ranking task.

To overcome these problems, Google implemented purpose-built systems – MapReduce [DG04] for task (*i*) and Pregel [MAB+10] for task (*ii*). To address the first problem,

[1] Slightly varying from standard mathematical notation, we write the projection functions associated with the product type components (such as `creditType`) inlined in the product type definition.

these systems were designed to scale out to thousands of commodity hardware nodes in the presence of frequent failures. To address the second problem, each system adopted a parallel dataflow graph with a fixed shape that was suitable for the targeted task. Instead of SQL, the dataflows were constructed in a general-purpose programming environment such as C++, using UDFs and UDTs.

The impact of these systems was twofold. On the one hand, they triggered the development of open-source projects that re-implemented the proposed designs and programming models – Apache Hadoop (for MapReduce), and Apache Giraph (for Pregel). On the other, they spurred the interest of the data management and distributed systems research communities, where much of the ideas presented below originated.

2.3.2 Spark RDD and Flink DataSet

MapReduce and Pregel allowed users to process data flexibly and at a scale that was not possible with traditional data management solutions. At the same time, 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 [ZCF+10] and Nephele/PACTs [BEH+10] (which became Stratosphere and then Flink) were designed to overcome these limitations.

Generalizing MapReduce, these systems were able to execute dataflow graphs freely composed 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 offered type-based DSLs deeply embedded in JVM-based GPLs like Scala or Java. This technique was used from the onset by Spark, and later also adopted by Stratosphere/Flink [Har13]. Both eDSLs are based on a generic type representing a distributed, unordered collection of homogeneous elements with duplicates. In Spark, the type is called `RDD` (short for *Resilient Distributed Dataset*), while in Flink the type is called `DataSet`.

The `RDD` and `DataSet` eDSLs represent a significant improvement over the imperative style of dataflow assembly employed by Hadoop's MapReduce or Stratosphere's PACTs APIs. Nevertheless, a closer look reveals a number of important limitations shared between both eDSLs. To illustrate those, we use a series of examples using a simplified film database schema[2].

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

[2] Product (or *struct*) types can be encoded as case classes in Scala and used as data model in both eDSLs.

Example 2.1 (Operator Chains)**.** To demonstrate the similarity between the two eDSLs, consider the following Scala code snippet, which filters movies from the 1990s and projects their year and name. Modulo the 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 the above code in Scala will append a chain of a `filter` (1), a `map` (2), and a `filter` (3) operator to the dataflow graph associated with `movies` and wrap the result in a new `RDD`/`DataSet` instance bound to `titles`. While this functional (or algebraic) style of dataflow assembly is concise and elegant, it is not really declarative and optimizable. To see why, compare the above code 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 both selection predicates behind the projection. For the dataflow graph discussed above, however, swapping (2) with (3) also implies adapting the function passed to (3), as the element type changes from (`Short`, `String`) to `Movie`. Since both eDSLs treat functions bound to second-order operators as "black-box" objects, the same rewrite cannot be realized directly in their IRs. To implement those, one has to resort to bytecode-level analysis and manipulation [HPS+12].

Example 2.2 (Join Cascades)**.** For the next example, consider a code fragment that joins movies with people over credits.

```
// 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 these code snippets. First, a standard, declarative syntax like *Select-From-Where* in SQL is not available. Instead, n-ary joins must be specified as a DSL-specific cascade of binary `join` operators. Consequently, the element type in the result is a tuple of nested pairs whose shape reflects the shape of the join tree. For example, the type of `ys` is ((`Movie`, `Credit`), `Person`). Field access therefore requires projection chains that traverse the nested tuple tree to its leafs. For example,

projecting (movie title, person name) pairs from `ys` can be done in one of two ways.

```
// total function
// with explicit 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. Due to the same reason stated in Example 2.1 (black-box function parameters), neither of these optimizations is possible in the discussed eDSLs. Solutions proposed in the past [GFC+12, HPS+12] indicate the potential benefits of such optimizations, but rely on an auxiliary bytecode inspection or bytecode de-compilation step.

Example 2.3 (Reducers)**.** Computing global or per-group aggregates is an integral operation in most data analysis pipelines. MapReduce is a powerful model for computing User-Defined Aggregates (UDAs) in parallel. Here is how one can count the total number of movies using `map` and `reduce` in Spark's `RDD` and Flink's `DataSet` APIs.

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

And here is how one can to count 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 a specific parallel execution strategy. The input values (or the values of each group) are thereby reduced to a single aggregate value (or one aggregate per group) in parallel by means of repeated application of an associative and commutative binary function specified by the programmer. Aggressive use of reducers can therefore improve performance and scalability of the evaluated dataflows.

Optimal usage patterns, however, can be hard to identify, especially without a good background in functional programming. For example, in order to check whether Alfred Hitchcock or Woody Allen has directed more movies, one might build upon the `ys` collection of ((`Movie`, `Credit`), `Person`) triples defined in Example 2.2.

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

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

```
// pair-count movies directed by (Alfred Hitchcock, Woody Allen)
val (c1, c2) = ys
  .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))
// compare the two counts
c1 < c2
```

Another pitfall arises when handling groups. As group values cannot always be processed by an associative and commutative function, the discussed eDSLs offer alternative operators for "holistic" group processing. These operators apply the function parameter once per group, exposing all group values as a Scala `Iterator` (Flink) or `Iterable` (Spark). For example, the number of movies per decade can also be counted 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)
  })
```

Understanding group processing in terms of a `groupBy` and a `map` over each group is more convenient than in terms of a `map` followed by a `reduce` or `reduceByKey`. However, a common mistake is to encode dataflows in the former style even if they can be defined in the latter. Grouping requires data re-partitioning, and in the case of a subsequent `reduce` the amount of shuffled data can be significantly reduced by pushing partial `reduce` computations before the shuffle step. Flink fuses a `groupBy` followed by a `reduce` implicitly, while Spark requires a dedicated operator called `reduceByKey`.

As with the previous two examples, optimizing these cases by means of automatic term rewriting is not possible in the presented eDSLs. Constructing efficient dataflows is predicated on the programmer's understanding of the operational semantics of operators like `reduce` and `reduceByKey`.

Example 2.4 (Caching)**.** Dataflow graphs constructed by `RDD` and `DataSet` terms might be related by the enclosing data- and control-flow structure[3]. For example, the `ys` collection from Example 2.2 is referenced twice in the naïve "compare movie-counts" implementation from Example 2.3 – once when counting the movies of Hitchcock (`c1`) and once when counting the movies of Allen (`c2`). Since a global `reduce` implicitly triggers evaluation, the dataflow graph associated with `ys` is expanded and evaluated twice. To amortize the evaluation cost of the shared sub-graph, the `RDD` eDSL offers a dedicated `cache` operator.

```
// cache shared sub-graph
val us = ys
  .filter(_._1._2.creditType == "director")
  .cache()
// count movies directed by Alfred Hitchcock
val c1 = us
  .map(y => if (y._2.name == "Hitchcock, Alfred") 1L else 0L)
  .reduce((u, v) => u + v)
// count movies directed by Woody Allen
val c2 = us
  .map(y => if (y._2.name == "Allen, Woody") 1L else 0L)
  .reduce((u, v) => u + v)
// compare the two counts
c1 < c2
```

Although Flink currently lacks first-class support for caching, a `cache` operator can be defined in terms of a pair of `write` and `read` operators and used with similar effect.

Data caching can also 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.iterate(N) ( w =>
  update(S, w)
)
```

[3] We use the spelling *dataflow* to denote bulk collection processing programs, and *data-flow* to denote the def-use relation between value bindings in the sense used in the language compilation literature.

The Spark version requires two explicit `cache` calls. If we do not call `cache` on the `static` result, the `static` dataflow graph will be evaluated N times. If we do not call `cache` on the `update` result, the loop body will be replicated N times without enforcing evaluation. As in the previous case, the Flink optimizer can automatically decide which dataflow graphs are loop-invariant and can be cached. However, in order to do this, the `DataSet` eDSL enforces the use of a dedicated `iterate` operator which models a restricted class of control-flow structures.

To summarize, Spark reuses Scala control-flow constructs, but delegates decisions about caching to the programmer. Flink, on the other hand, can often optimize some of these decisions automatically, but to achieve this it requires dedicated (and restricted) control-flow primitives and thereby violates the *"linguistic reuse"* design principle.

2.3.3 Current Solutions

To address the problems outlined above, two solution approaches are currently pursued. The first approach is to fall back to stand-alone DSLs. Notable sDSLs in this category are Pig Latin [ORS+08], Hive [TSJ+09], and SparkSQL [AXL+15]. This approach allows for both declarative syntax and advanced optimizations, as the entire AST of the input program can be considered in the compiler pipeline. 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 "lift" lambda expressions passed to second-order from "black-box" host-language constructs to first-class eDSL citizens. Notable examples in this category are `DataFrame` and `Dataset` eDSLs in Spark [AXL+15] and the `Table` eDSL in Flink [Kre15]. The benefit of this approach is that filter, selection, and grouping expressions are represented in the IR. This enables logical optimizations such as join reordering, filter and selection push-down, and automatic use of partial aggregates. The problem is that by "lifting" the expression language one loses the ability to reuse host-language syntax for anonymous function declaration, field projections, and arithmetic operators and types. The embedding strategy of state-of-the-art solutions is based either on plain strings or on a dedicated type (`Expression` in Flink, `Column` in Spark). The linguistic reuse principle is violated in both cases. The following examples illustrate the result with a simple select-and-project dataflow.

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

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

Neither of these approaches benefits from the type-safety or syntax checking capabilities of the host language. For example, the `filter` expression in the string-based approach is syntactically incorrect, as it lacks the closing quote after `director`, and the type-based versions have `creditType` misspelled. However, the enclosing Scala programs will compile silently, and the errors will be caught only at runtime, once the eDSL attempts to evaluate the resulting dataflows. In situations where long-running, possibly iterative computations are aborted at the very end due to a typing error, these issues can be particularly frustrating to the programmer. As an additional source of confusion, `filter` is overloaded to accept (black-box) regular Scala lambdas next to the reflected, but more idiosyncratic DSL-specific expressions, and, similarly, one can use `map` instead of `select`. Why and when should we prefer one variant over the other? Which expressions can be specified in the embedded language and which cannot? As with the eDSLs discussed in Section 2.3.2, the burden of understanding and navigating these trade-offs is on the programmer.

3 Solution Approach

Section 2.3 outlined a number of limitations shared between state-of-the-art DSLs for parallel dataflow systems such as `DataSet` and `RDD` as well as problems with existing solutions. To identify the root cause of these problems, we have to position these DSLs in the design space from Figure 2.1. Observe that the embedding strategy is *based on types*. Because of this, the IR lifted by DSL terms can only reflect method calls on these types as well as their def-use relation. In Section 2.3, the source code fragments reflected in the IR were highlighted in a different color. The remaining syntax (printed in black) is not reflected at the IR level. This includes the "glue code" connecting dataflow definitions, as well as lambdas passed as operator arguments.

The implications of this design decision for the optimizability, linguistic reuse, and declarativity are severe. In Example 2.1, it prohibits automatic operator reordering. In Example 2.2, it prohibits automatic join-order optimization as well as the use of `for`-comprehensions – a standard, declarative syntax for `Select-From-Where`-style expressions available in Scala. In Example 2.3, it prohibits automatic use of partial aggregates. In Example 2.4, it either prohibits automatic selection of optimal caching strategies or violates the linguistic reuse principle.

The net effect are eDSLs which on the surface seem straightforward to use, yet for most applications require some degree of expert knowledge in data management and distributed systems in order to produce fast and scalable programs. The appeal of declarative, yet performant bulk dataflow languages such as SQL is lost.

As a solution to these problems, we propose a design for a quotation-based DSL for parallel collection processing embedded in Scala. Utilizing Scala's reflection capabilities, this approach allows for deeper integration with the host language. In line with the objectives from Section 2.2, this leads to improved linguistic reuse and reduced linguistic noise. At the same time, a more principled collection processing API allows for optimizing transformations targeting the type-based eDSLs presented above. This results in a language for scalable data analysis where notions of data-parallel computation no longer

leak through the core programming abstractions. Instead, parallelism becomes *implicit* for the programmer without incurring significant performance penalty.

To illustrate the main difference between the proposed quotation-based approach against and state-of-the-art type-based embedding, compare the `RDD`-based movies-per-decade example from Example 2.3 against a code snippet expressed in the *Emma* API wrapped in an `onSpark` quote.

```
// RDD (Spark)

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

```
// Our solution (with Spark backend)
onSpark {
  for {
    g <- movies.groupBy(decade(_.year))
  } yield (g.key, g.values.size)
}
```

Observe how with the quotation-based approach, we use the more intuitive `groupBy` followed by a map over each group using a Scala `for`-comprehension. The quoted code snippet is highlighted, indicating that we can reflect everything in the IR of our eDSL. This allows us to (*i*) inspect all uses of `g.values` (in this case, only `g.values.size`), (*ii*) determine that `size` can be expressed in terms of partial aggregates, and (*iii*) automatically generate code which executes the program on Spark using the `reduceByKey` primitive from the `RDD`-based snippet.

4 Background

This section gives methodological background relevant to our solution approach. Section 4.1 outlines a category-theoretic foundation for distributed collections and parallel collection processing based on Algebraic Data Types (ADTs), structural recursion, and monads, introducing these concepts from first principles. Section 4.2 reviews IRs common in the compiler community – Static Single Assignment (SSA) form and a functional encoding of SSA called Administrative Normal Form (ANF).

4.1 Category Theory

Category theory can be used as a framework for modeling various subjects of study in a concise mathematical way. We use category theory to set up a constructive model for distributed collections and parallel collection processing, highlighting the connection between some theorems associated with the categorical constructions and the corresponding optimizations for parallel collection processing workloads.

The development in this section is restricted to definitions and constructions relevant to the subject of this thesis. Pierce gives a general introduction to category theory with focus on computer science applications [Pie91]. Bird and de Moor offer a more detailed treatment with focus on calculational program reasoning [BdM97]. Ehrig and Mahr outline a categorical view of algebraic specifications based on initial semantics [EM85]. Wadler [Wad92] gives a detailed introduction to monads and monad comprehensions. Chapter 2 in [Gru99] uses categorical collection types and monads as a basis for the development of a functional IR for database queries. Here, we essentially recast a subset of the theory presented in [Gru99] as a formalism which explains the optimizations outlined in Section 2.3 and therefore guides the design of the user-facing API (in Chapter 5) and IR (in Chapter 6) of the proposed embedded DSL. The equational rewrites in this thesis are carried out using the so-called Bird-Meertens formalism [Bac88, Gib94] also adopted in [BdM97, Gru99].

4.1.1 Basic Constructions

Category. A *category* $\mathbf{C} = (\mathrm{Ob}_{\mathbf{C}}, \mathrm{Mor}_{\mathbf{C}}, \circ, id)$ is a mathematical structure consisting of the following components.

- A class of *objects* $A, B, C, \ldots \in \mathrm{Ob}_{\mathbf{C}}$.
- A set of *morphisms* $\mathrm{Mor}_{\mathbf{C}}(A, B)$ for each pair of objects $A, B \in \mathrm{Ob}_{\mathbf{C}}$. Each morphism can be seen as a unique $\mathbf{C}$-arrow $f : A \to_{\mathbf{C}} B$ connecting the source $src\, f = A$ and target $tgt\, f = B$ objects of the underlying set $\mathrm{Mor}_{\mathbf{C}}(A, B)$. We omit the subscript $\mathbf{C}$ and write $f : A \to B$ when the underlying category $\mathbf{C}$ is clear from the context.
- A composition operator

  $$\circ : \mathrm{Mor}_{\mathbf{C}}(A, B) \times \mathrm{Mor}_{\mathbf{C}}(B, C) \to \mathrm{Mor}_{\mathbf{C}}(A, C)$$

  which maps pairs of morphisms with a matching "apex" object to their composition

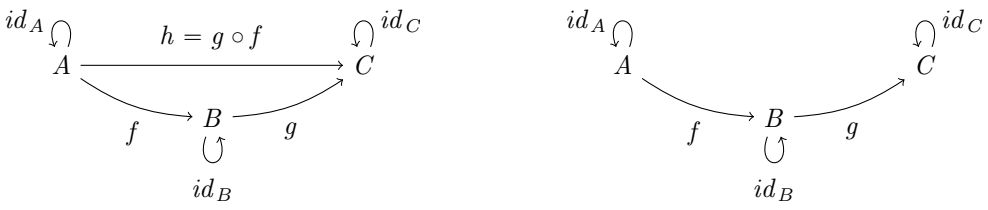
  $$(f, g) \mapsto g \circ f \quad .$$

- A family of *identity morphisms* $id_A \in \mathrm{Mor}_{\mathbf{C}}(A, A)$ for all $A \in \mathrm{Ob}_{\mathbf{C}}$.

In addition, a category $\mathbf{C}$ satisfies the following *associativity* and *identity* axioms for all $A, B, C, D \in \mathrm{Ob}_{\mathbf{C}}$, $f \in \mathrm{Mor}_{\mathbf{C}}(A, B)$, $g \in \mathrm{Mor}_{\mathbf{C}}(B, C)$, and $h \in \mathrm{Mor}_{\mathbf{C}}(C, D)$.

$$\begin{gathered} (h \circ g) \circ f = h \circ (g \circ f) \\ f \circ id_A = f \quad \& \quad id_B \circ f = f \end{gathered} \tag{Category}$$

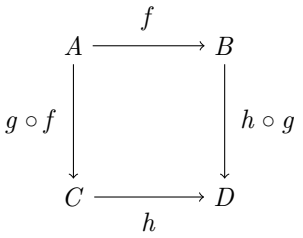
Categories can be represented visually as directed multi-graphs whose nodes correspond to objects and whose edges correspond to morphisms. Note that not all directed multi-graphs constitute a valid category. For example, from the following two graphs

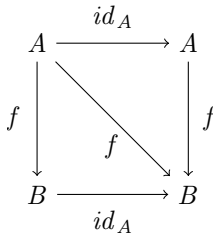


the left one corresponds to a category with three nodes and six morphisms, while the right one does not, as it lacks an $A \to C$ edge corresponding to the $g \circ f$ morphism.

Similarly, the Category axioms can be represented as commutative diagrams. Stating that the left- and right-hand sides of the Category equations must be equal is the same as stating that the square (on the left) and the two triangles (on the right) of the

following two diagrams must commute.





For the purposes of this thesis, our focus lies primarily in the category **Set**. Objects like $A, B \in \mathrm{Ob}_{\mathbf{Set}}$ denote *types*, while morphisms like $f \in \mathrm{Mor}_{\mathbf{Set}}(A, B)$ and $g \in \mathrm{Mor}_{\mathbf{Set}}(B, C)$ denote *total functions* with corresponding domain and codomain types given respectively by $src\,\cdot$ and $tgt\,\cdot$. The identity morphisms $id_A \in \mathrm{Mor}_{\mathbf{Set}}(A, A)$ denote identity functions $\forall a \in A.\ a \mapsto a$ and the composition operator denotes function composition $\forall a \in A.\ (f \circ h)\, a = f\,(h\, a)$. The validity of the Category axioms follows immediately from these definitions and the associativity of function application.

While conceptually thrifty, the language of category theory is surprisingly expressive. For example, a concept that can be generalized from **Set** to an arbitrary category **C** and defined in pure categorical terms is the notion of isomorphism.

Isomorphism. An *isomorphism* is a morphism $f \in \mathrm{Mor}_{\mathbf{C}}(A, B)$ with a corresponding inverse morphism. That is, there exists some $g \in \mathrm{Mor}_{\mathbf{C}}(B, A)$, sometimes denoted f^{-1}, such that the following equations hold.

$$f \circ g = id_B \quad \text{and} \quad g \circ f = id_A \qquad \text{(Isomorphism)}$$

Two objects A and B related by an isomorphism are said to be isomorph, written $A \cong B$.

The simplest kind of categorical constructions relate objects and morphisms within the same category. Each construction is defined as the solution of an associated family of equations (sometimes also called *universal properties*). The solution can be shown to be unique up to isomorphism. An interesting property of most constructions is that by reversing the direction of all morphisms one can obtain an associated *dual* construction. We now introduce two pairs of dual constructions and discuss their interpretation in **Set**.

Initial Object. An object $0 \in \mathrm{Ob}_{\mathbf{C}}$ is called *initial* in **C** if, for every object $A \in \mathrm{Ob}_{\mathbf{C}}$, the set $\mathrm{Mor}_{\mathbf{C}}(0, A)$ consists of exactly one morphism, denoted $!_A$.

Final Object. An object $1 \in \mathrm{Ob}_{\mathbf{C}}$ is called *final* in **C** if, for every object $A \in \mathrm{Ob}_{\mathbf{C}}$, the set $\mathrm{Mor}_{\mathbf{C}}(1, F)$ consists of exactly one morphism, denoted ¡$_A$.

The diagrams corresponding to these definitions look as follows (dashed lines indicate that the associated morphism is unique).

To illustrate the flavor of categorical proofs, we show that, if they exist, all initial objects in a category are unique up to isomorphism. Suppose that two objects A and B are both initial in $\mathbf{C}$. Then, from Initial Object applied to A, it follows that $\mathrm{Mor}_{\mathbf{C}}(A, B) = \{\, !_B \,\}$ and $\mathrm{Mor}_{\mathbf{C}}(A, A) = \{\, id_A \,\}$. Similarly, from Initial Object applied to B, it follows that $\mathrm{Mor}_{\mathbf{C}}(B, A) = \{\, !_A \,\}$ and $\mathrm{Mor}_{\mathbf{C}}(B, B) = \{\, id_B \,\}$. Since $\mathbf{C}$ is a category, the compositions $!_A \circ !_B \in \mathrm{Mor}_{\mathbf{C}}(A, A)$ and $!_B \circ !_A \in \mathrm{Mor}_{\mathbf{C}}(B, B)$ must also exist, and the only options we have are $!_A \circ !_B = id_A$ and $!_A \circ !_B = id_B$, asserting that A and B are indeed isomorph. A similar proof for terminal objects follows along the same line of reasoning.

The initial object in **Set** is the *empty set* $\varnothing$, and the initial morphisms are the empty functions $\varnothing \to A$. Dually, the final object in **Set** is the *singleton set* $\{\,()\,\}$ (that is, the set of one element), and the final morphisms $A \to \{\,()\,\}$ are the constant functions $\forall a \in A.\ a \mapsto ()$. If **Set** is viewed from a type-theoretic perspective, the initial object corresponds to the *bottom type* (called `Nothing` in Scala), and the terminal object to the *unit type* (called `Unit` in Scala). Note also that in **Set** we have $A \cong \mathrm{Mor}_{\mathbf{Set}}(1, A)$ – elements $a \in A$ are in one-to-one correspondence with the constant functions $() \mapsto a$.

The next definitions allow us to construct objects and morphisms out of existing ones.

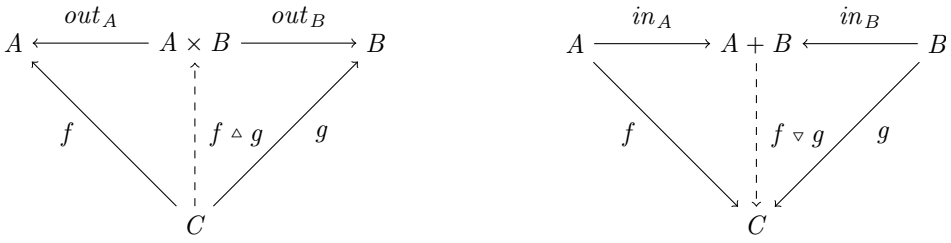
Product. Given a pair of objects A and B in $\mathbf{C}$, their *product*, denoted $A \times B$, is an object in $\mathbf{C}$ with associated *projection morphisms* $out_A : A \times B \to A$ and $out_B : A \times B \to B$ which satisfies the following universal property. For every object $C \in \mathbf{C}$ with morphisms $f : C \to A$ and $g : C \to B$ there exists a unique morphism, denoted $f \vartriangle g$, such that the following equations hold.

$$out_A \circ (f \vartriangle g) = f \quad \text{and} \quad out_B \circ (f \vartriangle g) = g \tag{Product}$$

Coproduct. Given a pair of objects A and B in $\mathbf{C}$, their *coproduct*, denoted $A + B$, is an object in $\mathbf{C}$ with associated *injection morphisms* $in_A : A \to A \times B$ and $in_B : B \to A \times B$ which satisfies the following universal property. For every object $C \in \mathbf{C}$ with morphisms $f : A \to C$ and $g : B \to C$ there exists a unique morphism, denoted $f \triangledown g$, such that the following equations hold.

$$(f \triangledown g) \circ in_A = f \quad \text{and} \quad (f \triangledown g) \circ in_B = g \tag{Coproduct}$$

Again, if they exist, the product and coproduct objects can be shown to be unique up to isomorphism. The diagrams associated with these two definitions look as follows.



As a corollary, we obtain two laws which allow us to fuse a morphism h with a subsequent product morphism or preceding coproduct morphism.

$$(f \vartriangle g) \circ h = (f \circ h) \vartriangle (g \circ h) \qquad \text{(Product-Fusion)}$$
$$h \circ (f \triangledown g) = (h \circ f) \triangledown (h \circ g) \qquad \text{(Coproduct-Fusion)}$$

In **Set**, the product $A \times B$ corresponds to the *cartesian product* of the sets A and B, and the corpoduct $A + B$ to their *tagged union*. From a type-theoretic perspective the constructions in **Set** can be interpreted as *product* and *sum* types, and the corresponding universal morphisms as the following functions.

$$\forall c \in C.\quad (f \vartriangle g)\, c = (f\, c, g\, c)$$
$$\forall x \in A + B.\quad (f \triangledown g)\, x = \begin{cases} f\, x & \text{if } x \in A \\ g\, x & \text{if } x \in B \end{cases}$$

Conceptually, products offer a categorical notion of *delineation* – we describe an action (morphism) into a product object $A \times B$ component-wise, by individually describing the actions for each possible part (A and B). Dually, coproducts offer a categorical notion of *lineage* – we describe an action (morphism) out of a coproduct object $A + B$ component-wise, by individually describing the actions for each possible case (A or B).

Binary products and coproducts are commutative and associative up to isomorphism. The Coproduct and Product definitions therefore can be generalized to n-ary products and coproducts, denoted respectively $\prod_{i=1}^{n} A_i$ and $\coprod_{i=1}^{n} A_i$.

Finally, we introduce the notion of a product category which is needed for the generalization of products and coproducts as functors in the next section.

Product Category. For any pair of categories $\mathbf{C}$ and $\mathbf{D}$, the *product category* $\mathbf{C} \times \mathbf{D}$ has as objects pairs (A, B) where $A \in \mathrm{Ob}_{\mathbf{C}}$ and $B \in \mathrm{Ob}_{\mathbf{D}}$, and as morphisms pairs (f, g) where $f \in \mathrm{Mor}(C)$ and $g \in \mathrm{Mor}(D)$. Morphism composition and identity are defined pairwise: $(g, f) \circ (i, h) = (g \circ i, f \circ h)$ and $id_{(A,B)} = (id_A, id_B)$.

4.1.2 Functors

So far, our categorical vocabulary has been restricted to constructions within a single category. As a next step, we focus on *constructions between categories*. The most basic case is a mapping between categories that preserves the structure of the source category.

Functor. Given two categories, $\mathbf{C}$ and $\mathbf{D}$, a *functor* $\mathsf{F} = (\mathsf{F}_{\mathrm{Ob}}, \mathsf{F}_{\mathrm{Mor}})$ is a mapping from $\mathbf{C}$ to $\mathbf{D}$ consisting of a component

$$\mathsf{F}_{\mathrm{Ob}} : \mathrm{Ob}_{\mathbf{C}} \to \mathrm{Ob}_{\mathbf{D}}$$

that operates on objects, and a component

$$\mathsf{F}_{\mathrm{Mor}} : \mathrm{Mor}_{\mathbf{C}}(A, B) \to \mathrm{Mor}_{\mathbf{C}}(\mathsf{F}_{\mathrm{Ob}}(A), \mathsf{F}_{\mathrm{Ob}}(B))$$

that operates on morphisms, preserving identity and composition:

$$\mathsf{F}id_A = id_{\mathsf{F}A} \quad \text{and} \quad \mathsf{F}(g \circ f) = \mathsf{F}g \circ \mathsf{F}f \ . \tag{Functor}$$

To simplify notation, we omit the component subscript and write $\mathsf{F}A$ instead of $\mathsf{F}_{\mathrm{Ob}}(A)$ and $\mathsf{F}f$ instead of $\mathsf{F}_{\mathrm{Mor}}(f)$.

An *endofunctor* is a functor whose source and target categories coincide. From a type-theoretic perspective, **Set** endofunctors encode the notion of *universal polymorphism*. For example, the type of lists with elements of type A, usually written $\forall A.\, \mathsf{List}A$, can also be seen as a $\mathbf{Set} \to \mathbf{Set}$ functor $A \mapsto \mathsf{List}A$ that maps an element type A to its corresponding list type $\mathsf{List}A$. In a similar way, collection types such as Bag and Set can also be understood as functors. With the definitions so far, however, the internals of these functors are "black-box". For a "white-box" view, we have to formalize the notion of an Algebraic Data Type (ADT) in a categorical setting. Do achieve this, we start by introducing a number of base functors.

Identity Functor. The identity functor $\mathsf{Id} : \mathbf{C} \to \mathbf{C}$ maps objects and morphisms to themselves: $\mathsf{Id}A = A$ and $\mathsf{Id}f = f$.

Constant Functor. The constant functor $\mathsf{K}_A : \mathbf{C} \to \mathbf{D}$ maps $\mathbf{C}$-objects to a fixed $\mathbf{D}$-object A and morphisms to id_A, i.e. $\mathsf{K}_A B = A$ and $\mathsf{K}_A f = id_A$.

Assuming that products and coproducts exist for arbitrary A and B in $\mathrm{Ob}_{\mathbf{C}}$, we can define corresponding $\mathbf{C} \times \mathbf{C} \to \mathbf{C}$ functors.

Product Functor. Let $\mathbf{C}$ be a category with products. Then the *product functor* $\cdot \times \cdot$ is a $\mathbf{C} \times \mathbf{C} \to \mathbf{C}$ functor defined as follows. For any two $\mathbf{C}$-objects A and B, the functor mapping is their product construction $(A, B) \mapsto A \times B$. Similarly, for any two morphisms $f : A \to B$ and $g : C \to D$, the functor mapping $f \times g : A \times C \to B \times D$ is defined as

$f \times g = (f \circ out_A) \vartriangle (g \circ out_B)$.

Coproduct Functor. Let $\mathbf{C}$ be a category with coproducts. Then the *coproduct functor* $\cdot + \cdot$ is a $\mathbf{C} \times \mathbf{C} \to \mathbf{C}$ functor defined as follows. For any two $\mathbf{C}$-objects A and B, the functor mapping is their coproduct construction $(A, B) \mapsto A + B$. Similarly, for any two morphisms $f : A \to B$ and $g : C \to D$, the functor mapping $f + g : A + C \to B + D$ is defined as $f + g = (in_B \circ f) \triangledown (in_D \circ g)$.

As a corollary, we obtain laws which enable fusing a functor mapping of a pair of morphisms (h, i) with a preceding product morphism or subsequent coproduct morphism.

$$(h \times i) \circ (f \vartriangle g) = (h \circ f) \vartriangle (i \circ g) \qquad \text{(Product-Functor-Fusion)}$$
$$(f \triangledown g) \circ (h + i) = (f \circ h) \triangledown (g \circ i) \qquad \text{(Coproduct-Functor-Fusion)}$$

Functors are closed under composition – if F and G are functors, so is $\mathsf{GF}\cdot = \mathsf{G}(\mathsf{F}\cdot)$. Functors composed from Id, K_A, $\times$, and $+$ are called *polynomial functors*. Polynomial functors are closely related to the concept of F-algebras.

4.1.3 F-Algebras

F-algebra. Let F denote an endofunctor in a category $\mathbf{C}$. An F-algebra $\alpha : \mathsf{F}A \to A$ is a morphism in $\mathbf{C}$. The functor F is called *signature* or *base* functor, and the object A is called *carrier* of α.

F-algebras provide a compact framework for modeling terms of type A. If F is polynomial, its general form $\mathsf{F}A = \coprod_{i=1}^{n} X_i$ implies that α factors into a family of morphisms $\alpha_i : X_i \to A$. For **Set**-valued functors, this factorization can be seen as an encoding of a polymorphic interface consisting of n functions α_i with shared, generic return type A.

As an example, consider F-algebras for the **Set** endofunctor $\mathsf{F} = \mathsf{K}_1 + \mathsf{Id}$ which maps A to $1 + A$. In this case, F-algebras are functions $\alpha : 1 + A \to A$ with carrier type A. From the Coproduct universal property, we know that α can be factored as $zero \triangledown succ$, where $zero = \alpha \circ in_1 : 1 \to A$ and $succ = \alpha \circ in_A : A \to A$. For a fixed type A, every possible combination of suitable $zero$ and $succ$ functions gives rise to a different F-algebra. The following lines list three F-algebras.

$$\alpha = zero_\alpha \triangledown succ_\alpha : \mathsf{F}\mathbb{Z} \to \mathbb{Z} \qquad zero_\alpha\,() = 0 \qquad succ_\alpha\, x = x + 1$$
$$\beta = zero_\beta \triangledown succ_\beta : \mathsf{F}\mathbb{Z} \to \mathbb{Z} \qquad zero_\beta\,() = 0 \qquad succ_\beta\, x = x - 1$$
$$\gamma = zero_\gamma \triangledown succ_\gamma : \mathsf{F}\mathbb{C} \to \mathbb{C} \qquad zero_\gamma\,() = 0 \qquad succ_\gamma\, x = x^2 - 1$$

For each $\chi \in \{\alpha, \beta, \gamma\}$, we can then compose $zero_\chi$ and $succ_\chi$ in order to build terms of the corresponding carrier type. Terms with the general form ${succ_\alpha}^n \circ zero_\alpha$ for $n > 0$

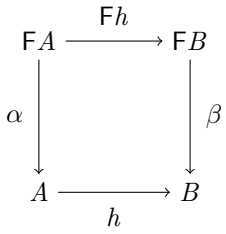
correspond to positive integers, $succ_\beta{}^n \circ zero_\beta$ terms correspond to negative integers, and $succ_\gamma{}^n \circ zero_\gamma$ terms correspond to members of the sequence $P_c^n(0)$ of iterated applications of the complex polynomial[1] $P_c : x \mapsto x^2 + c$ for $c = -1$.

As a next step, consider carrier morphisms preserving the structure of F-algebra terms.

F-homomorphism. Fix two F-algebras $\alpha : \mathsf{F}A \to A$ and $\beta : \mathsf{F}B \to B$. An F-homomorphism is a $\mathbf{C}$-morphism $h : A \to B$ satisfying the equation

$$h \circ \alpha = \beta \circ \mathsf{F}h \tag{F-Hom}$$

which is also represented by the following commutative diagram.



As before, if F is a polynomial functor in **Set**, the above definition has a more specific interpretation. Informally, in this case F-Hom is states that applying h on the result of α_i is the same as applying h on the A-arguments of α_i and then applying β_i instead.

In our running example where $\mathsf{F} = \mathsf{K}_1 + \mathsf{Id}$, F-Hom states that $h : A \to B$ is an F-homomorphism between any two F-algebras (A, α) and (B, β) if and only if

$$h\, zero_\alpha = zero_\beta \quad \text{and} \quad h\,(succ_\alpha\, a) = succ_\beta\,(h\, a)$$

for all $a \in A$. For example, $h : \mathbb{Z} \to \mathbb{Z}$ with $h\, z = -z$ is an F-homomorphism between α and β, verified as follows.

$$h\, zero_\alpha = 0 = zero_\beta$$
$$h\,(succ_\alpha\, a) = h\,(a+1) = -(a+1) = -a - 1 = h\, a - 1 = succ_\beta\,(h\, a)$$

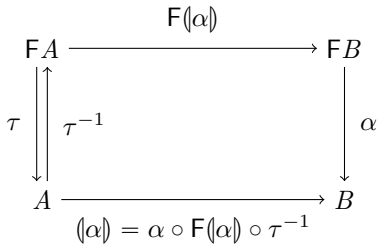
F-homomorphisms preserve identity morphisms and are closed under composition. F-algebras (as objects) and F-homomorphisms (as morphisms) thereby form a category denoted $\mathbf{Alg}(\mathsf{F})$. To understand the connection between F-algebras and ADTs, we fix F and consider initial objects in $\mathbf{Alg}(\mathsf{F})$. An initial object in $\mathbf{Alg}(\mathsf{F})$ is an F-algebra $\tau : \mathsf{F}T \to T$ such that each F-algebra $\alpha : \mathsf{F}A \to A$ induces a unique F-homomorphism between τ and α, denoted $(\!|\alpha|\!) : T \to A$. If τ is an isomorphism in $\mathbf{C}$, we can define $(\!|\alpha|\!)$

[1] The sequence $P_c^n(0)$ is used in the definition of the Mandelbrot set.

using the so-called *catamorphism* construction

$$(\!|\alpha|\!) = \alpha \circ \mathsf{F}(\!|\alpha|\!) \circ \tau^{-1} \tag{Cata}$$

as illustrated by the modified F-Hom diagram depicted below.



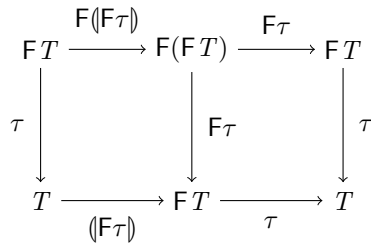
Verifying that Cata satisfies F-Hom (in other words, that the above diagram commutes) is a straight-forward consequence of the Isomorphism property of τ.

$$\begin{aligned}
& (\!|\alpha|\!) = \alpha \circ \mathsf{F}(\!|\alpha|\!) \circ \tau^{-1} \\
\Leftrightarrow \quad & \{ \text{ apply } \cdot \circ \tau \text{ on both sides } \} \\
& (\!|\alpha|\!) \circ \tau = \alpha \circ \mathsf{F}(\!|\alpha|\!) \circ \tau^{-1} \circ \tau \\
\Leftrightarrow \quad & \{ \text{ Isomorphism property of } \tau \} \\
& (\!|\alpha|\!) \circ \tau = \alpha \circ \mathsf{F}(\!|\alpha|\!)
\end{aligned}$$

Lambek's lemma [Lam93] asserts that the initial algebra τ exists, the induced unique homomorphisms always have the structure defined by Cata.

Lambek's Lemma. Let F be a $\mathbf{C}$-endofunctor such that $\mathbf{Alg}(\mathsf{F})$ has an initial object τ. Then the carrier T of τ and $\mathsf{F}\,T$ are isomorphic via τ.

Proof. To prove the above statement, we apply F to the initial F-algebra $\tau : \mathsf{F}\,T \to T$. The resulting $\mathbf{C}$-morphism $\mathsf{F}\tau : \mathsf{F}(\mathsf{F}\,T) \to \mathsf{F}\,T$ is also an F-algebra and, since τ is initial, there is a unique catamorphism $(\!|\mathsf{F}\tau|\!) : T \to \mathsf{F}\,T$. At the same time, τ can also be seen as an F-homomorphism between the F-algebras $\mathsf{F}\tau$ and τ. The above two observations are depicted by the following pair of commutative squares.



Since both $(\!|\mathsf{F}\tau|\!)$ and τ are F-homomorphisms, their composition $\tau \circ (\!|\mathsf{F}\tau|\!)$ must also be one.

But so is id_T and from the uniqueness of initial morphisms it follows that $\tau \circ (\!|\mathsf{F}\tau|\!) = id_T$. In the other direction, to show that $(\!|\mathsf{F}\tau|\!) \circ \tau = id_{\mathsf{F}T}$, we argue as follows.

$$
\begin{aligned}
& \tau \circ (\!|\mathsf{F}\tau|\!) = id_T \\
\Leftrightarrow \quad & \{ \text{ apply } \mathsf{F} \text{ on both sides } \} \\
& \mathsf{F}(\tau \circ (\!|\mathsf{F}\tau|\!)) = \mathsf{F} id_T \\
\Leftrightarrow \quad & \{ \text{ FUNCTOR properties } \} \\
& \mathsf{F}\tau \circ \mathsf{F}(\!|\mathsf{F}\tau|\!) = id_{\mathsf{F}T} \\
\Leftrightarrow \quad & \{ \text{ F-HOM property of } (\!|\mathsf{F}\tau|\!) \} \\
& (\!|\mathsf{F}\tau|\!) \circ \tau = id_{\mathsf{F}T}
\end{aligned}
$$

□

The existence of an initial algebra in $\mathbf{Alg}(\mathsf{F})$ is ensured if F is a polynomial functor [MA86]. Furthermore, if the underlying category is **Set**, Lambek's Lemma is tantamount to saying that τ must be a bijective function. The carrier set of an initial algebra T in **Set** therefore has the following properties. First, T has *no junk*: for all $t \in T$ there exists some $x \in \mathsf{F}T$ such that $\tau\, x = t$ (because τ is surjective). Second, T has *no confusion*: for all $x, y \in \mathsf{F}T$, $\tau\, x = \tau\, y$ implies $x = y$ (because τ is injective). In this case, we can interpret the components $\tau_i : X_i \to T$ of τ as data constructors, and the action of τ as a specific data constructor application. In the reverse direction, τ^{-1} acts like a parser, returning the constructor application term associated with a specific data point.

To illustrate these concepts, let us revisit the three F-algebras listed for the example functor $\mathsf{F} = \mathsf{K}_1 + \mathsf{Id}$. In the case of α (β), the carrier $\mathbb{Z}$ has (i) junk because negative (positive) integers cannot be expressed in terms of applications of the algebra functions, and (ii) confusion since, for example, $succ_\alpha\, {-1} = 0 = zero_\alpha$ ($succ_\beta\, 1 = 0 = zero_\beta$). In the case of γ, the carrier has confusion because, for example, $succ_\gamma\, {-1} = 0 = succ_\gamma\, 1$. We can make the carrier sets of α and β initial by restricting them to $\mathbb{N}$ and $\mathbb{Z}^{\leqslant 0}$. Since initial objects are unique up to isomorphism, we can then assert that the F-homomorphism $h : \mathbb{N} \to \mathbb{Z}^{\leqslant 0}$, $h\, n = -n$ is a bijection.

As a straight-forward consequence of CATA and F-HOM, we obtain two useful properties.

$$
\begin{aligned}
(\!|\tau|\!) &= id_T && \text{(CATA-REFLECT)} \\
h \circ \alpha = \beta \circ \mathsf{F}h \quad &\Rightarrow \quad h \circ (\!|\alpha|\!) = (\!|\beta|\!) && \text{(CATA-FUSION)}
\end{aligned}
$$

We will make extensive use of those in calculational proofs carried out in the next sections.

The theory developed so far suffices to devise a constructive model for lists with fixed element type. To do that, consider the **Set**-endofunctor $\mathsf{F} = \mathsf{K}_1 + \mathsf{K}_{Int} \times \mathsf{Id}$. The carrier

of the initial algebra in $\mathbf{Alg}(\mathsf{F})$ is the type *ListInt* of lists with integer elements. The initial algebra itself decomposes into a pair of list constructors

$$emp : 1 \to ListInt \quad \text{and} \quad cons : Int \times ListInt \to ListInt \tag{ListInt-Ctor}$$

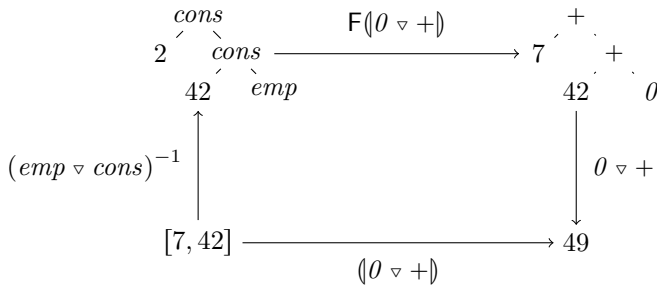
where $emp()$ denotes the empty list and $cons\, x\, xs$ the list constructed by inserting x in an existing list xs. Lists modeled by the above functor are therefore also called lists in *insert representation*. Each list instance is represented by a right-deep binary tree of *cons* applications terminating with a *emp* application. Catamorphisms in $\mathbf{Alg}(\mathsf{F})$ correspond to the functional *fold*. More specifically, for $\alpha = zero_\alpha \triangledown plus_\alpha : \mathsf{F}A \to A$ the catamorphism $(\!|\alpha|\!)$ corresponds to the function $fold\, \alpha$ which reduces *ListInt* values to a result of type A by means of structural recursion over its input.

$$\begin{aligned} (\!|\alpha|\!)\, (emp()) &= zero_\alpha() \\ (\!|\alpha|\!)\, (cons\, a\, as) &= plus_\alpha\, a\, ((\!|\alpha|\!)\, as) \end{aligned} \tag{ListInt-Fold}$$

To illustrate how Lambek's Lemma allows us to interpret structural recursion schemes such as ListInt-Fold in terms of their Cata components, consider applying the catamorphism

$$(\!|0 \triangledown +|\!) = (0 \triangledown +) \circ \mathsf{F}(\!|0 \triangledown +|\!) \circ (emp \triangledown cons)^{-1}$$

to a *ListInt* value $[2, 49]$. The $(emp \triangledown cons)^{-1}$ action deconstructs the input value, one layer at a time. The $\mathsf{F}(\!|0 \triangledown +|\!)$ action substitutes *emp* with *0* and *cons* with $+$ in the resulting parse tree, recursively calling $(\!|0 \triangledown +|\!)$ on all arguments of type *ListInt*. Finally, $0 \triangledown +$ evaluates the $(0 \triangledown +)$-algebra on the resulting tree, producing the final result 49. Expanding the recursive calls, these steps can be represented as follows.



4.1.4 Polymorphic Collection Types as Functors

In the previous section, we demonstrated how F-algebras for polynomial **Set**-endofunctors can be used to model collection types such as *ListInt*. The functor F encodes the signature of a family of base functions. The objects in $\mathbf{Alg}(\mathsf{F})$ represent all possible implementations (i.e., models) of this signature – a concept known as *classical semantics*. Finally, the carrier of the initial object and the associated catamorphisms in $\mathbf{Alg}(\mathsf{F})$ represent the

unique inductive (i.e., least fixpoint) type defined by F and its associated structural recursion scheme – a concept known as *initial semantics*.

The approach developed so far has two important limitations. First, the *ListInt* type is monomorphic (that is, with a fixed element type). Our goal, however, is to develop a polymorphic model where collection type constructors are understood as functors such as $\mathsf{List} : A \mapsto \mathsf{List}A$. Second, with the theory presented so far we can only model lists. This is at odds with the collection types exposed by systems such as Spark and Flink, where element order is not guaranteed. Formally, we want to model collections T where

$$cons\, a'\, (cons\, a\, as) = cons\, a\, (cons\, a'\, as)$$

for all $a, a' \in A, as \in \mathsf{T}A$. This implies that the associated carrier cannot be initial in the category $\mathbf{Alg}(\mathsf{K}_1 + \mathsf{K}_A \times \mathsf{Id})$ due to the introduced confusion. Consequently, we cannot define collection processing operations in terms of catamorphisms or derive optimizing program transformations from the catamorphism properties.

To overcome the first limitation, we generalize F-algebras to polymorphic F-algebras and use those to define the type functor $\mathsf{List} : A \mapsto \mathsf{List}A$. To overcome the second limitation, we extend signature functors F to specifications Spec and F-algebra categories $\mathbf{Alg}(\mathsf{F})$ to model categories $\mathbf{Mod}(\mathsf{Spec})$. Based on that, we define a hierarchy of collection type constructors List, Bag and Set in the so-called *insert representation*. In Section 4.1.5, we show how these type constructors can be defined in the so-called *union representation* using another signature functor G. Further, we discuss why G is a better fit for the distributed collections and dataflow frameworks presented in Section 2.3.

To simplify presentation, we restrict the definitions of an Algebraic Specification and Quotient F-Algebra to **Set**-endofunctors. This is sufficient for the goals of this thesis and allows for writing datatype equations in the the more familiar and readable point-wise form. A purely categorical, point-free development of datatype equations is proposed by Fokkinga [Fok92, Fok96] and adopted by Grust [GS99].

Polymorphic F-algebra. Let $\mathsf{F} : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$ be a functor and $\mathsf{F}_A : \mathbf{C} \to \mathbf{C}$ be defined by $\mathsf{F}_A\, B = \mathsf{F}(A, B)$ and $\mathsf{F}_A\, g = \mathsf{F}(id_A, g)$. A *polymorphic* F*-algebra* is a collection of F_A-algebras $\alpha_A : \mathsf{F}_A\, B \to B$ indexed by a *type parameter* $A \in \mathrm{Ob}_{\mathbf{C}}$. The F_A-algebras and F_A-homomorphisms for a fixed A form the category $\mathbf{Alg}(\mathsf{F}_A)$.

Polymorphic F-algebras allow us to abstract over the element type A in the collection types we want to model. For example, we can parameterize the signature functor for the *ListInt* algebra from $\mathsf{F}A = 1 + Int \times A$ to the following polymorphic signature.

$$\mathsf{F}(A, B) = 1 + A \times B \qquad \text{(Ins-Sign)}$$

Type Former. Let $\mathsf{F}_A : \mathbf{C} \to \mathbf{C}$ be a family of signature functors (or more generally

a family of specifications Spec_A) such that the associated family of model categories $\mathbf{Alg}(\mathsf{F}_A)$ (or more generally $\mathbf{Mod}(\mathsf{Spec}_A)$) has initial algebras τ_A for all $A \in \mathbf{C}$. The *type former* T associates each object $A \in \mathbf{C}$ with the carrier $\mathsf{T}A$ of τ_A.

Ultimately, we want to establish $\mathsf{T} \in \{\mathsf{List}, \mathsf{Bag}, \mathsf{Set}\}$ as type formers. To do so, we need to define specifications $\mathsf{Spec}_A^{\mathsf{T}}$ based on Ins-Sign, and construct the associated model categories $\mathbf{Mod}(\mathsf{Spec}_A^{\mathsf{T}})$. Assuming for the moment that this can be done, let

$$\forall A \in \mathbf{Set}. \quad \tau_A^{\mathsf{T}} = emp_A^{\mathsf{T}} \triangledown cons_A^{\mathsf{T}} : \mathsf{F}_A(\mathsf{T}A) \to \mathsf{T}A \tag{Ins-Ctor}$$

denote the polymorphic initial algebra associated with T. The structural recursion scheme ListInt-Fold generalizes to a family of definitions $(\!|\cdot|\!)_A^{\mathsf{T}}$ indexed by A. The catamorphism $(\!|\alpha|\!)_A^{\mathsf{T}}$ is the unique F_A-homomorphism in $\mathbf{Mod}(\mathsf{Spec}_A^{\mathsf{T}})$ between the F_A-algebras τ_A^{T} and some $\alpha = zero_\alpha \triangledown plus_\alpha : \mathsf{F}_A\, B \to B$, and is defined component-wise as follows.

$$\begin{aligned} (\!|\alpha|\!)_A^{\mathsf{T}}\,(emp_A^{\mathsf{T}}()) &= zero_\alpha() \\ (\!|\alpha|\!)_A^{\mathsf{T}}\,(cons_A^{\mathsf{T}}\,a\,as) &= plus_\alpha\,a\,((\!|\alpha|\!)_A^{\mathsf{T}}\,as) \end{aligned} \tag{Ins-Fold}$$

We have already established that for $\mathsf{T} = \mathsf{List}$ we can just use $\mathsf{Spec}_A^{\mathsf{T}} = \mathsf{F}_A$ and $\mathbf{Mod}(\mathsf{Spec}_A^{\mathsf{T}}) = \mathbf{Alg}(\mathsf{F}_A)$. Before discussing the corresponding choices when $\mathsf{T} \in \{\mathsf{Bag}, \mathsf{Set}\}$, we establish T as functors.

Type Functor. Let T be a type former for a polymorphic signature functor F. Then T can be extended to a *type functor* with object mapping inherited from the Type Former definition and morphism mapping defined as $\mathsf{T}f = (\!|\tau_B^{\mathsf{T}} \circ \mathsf{F}(f, id_{\mathsf{T}B})|\!)_A^{\mathsf{T}}$ for all $f : A \to B$.

The formal proof that T is indeed a functor can be found in [BdM97]. Here, we give an intuitive interpretation of $\mathsf{T}f$ when $\mathbf{C} = \mathbf{Set}$ and F is a polynomial functor.

First, observe that we can extend the F_B-algebra $\tau_B^{\mathsf{T}} : \mathsf{F}(B, \mathsf{T}B) \to \mathsf{T}B$ to an F_A-algebra $\alpha_A^{\mathsf{T}} : \mathsf{F}(A, \mathsf{T}B) \to \mathsf{T}B$ using composition in $\mathbf{C}$: $\alpha_A^{\mathsf{T}} = \tau_B^{\mathsf{T}} \circ \mathsf{F}(f, id_{\mathsf{T}B})$. In $\mathbf{Set}$, prepending $\mathsf{F}(f, id_{\mathsf{T}B})$ effectively adapts the type of all B-parameters to A. The adapted algebra converts arguments $a \in A$ to $fa \in B$ before passing them to the original τ_B^{T}-constructors.

For collection type functors T, the adapted algebra factors into $\alpha = emp_\alpha \triangledown cons_\alpha$, and the above statement can be formalized in terms of the Ins-Ctor definition of τ_B^{T}.

$$\begin{aligned} emp_\alpha &: 1 \to \mathsf{T}B && \text{defined as} && emp_\alpha() = emp_B^{\mathsf{T}}() \\ cons_\alpha &: A \times \mathsf{T}B \to \mathsf{T}B && \text{defined as} && cons_\alpha\,a\,bs = cons_B^{\mathsf{T}}\,(fa)\,bs \end{aligned}$$

We now have two F_A-algebras τ_A^{T} and α with carriers corresponding to the source and target objects of the morphism $f : A \to B$. Since τ_A^{T} is initial in $\mathbf{Mod}(\mathsf{Spec}_A^{\mathsf{T}})$, we can just set $\mathsf{T}f$ to be the catamorphism $(\!|\alpha|\!)_A^{\mathsf{T}} : \mathsf{T}A \to \mathsf{T}B$, which is also a $\mathbf{C}$-morphism.

If T is a collection type functor, we can plug-in the definitions of emp_α and $cons_\alpha$ into Ins-Fold, resulting in a definition of $\mathsf{T}f$ based on structural recursion.

$$\begin{aligned} \mathsf{T}f\,(emp_A^{\mathsf{T}}()) &= emp_B^{\mathsf{T}}() \\ \mathsf{T}f\,(cons_A^{\mathsf{T}}\,a\,as) &= cons_B^{\mathsf{T}}\,(fa)\,(\mathsf{T}f\,as) \end{aligned} \tag{Ins-Map}$$

Ins-Map reveals that the morphism mapping component of T corresponds to the higher-order function map^{T}, allowing us to use both notations interchangeably. The resulting $\mathsf{T}A \to \mathsf{T}B$ function maps input collection elements a to fa and returns the collection of mapped results. For example, $map^{\mathsf{List}}\,strlen : \mathsf{List}\,String \to \mathsf{List}\,Int$ emits the string length of each element in a list of strings, producing a list of integers.

$$map^{\mathsf{List}}\,strlen\,[Show, me, what, you, got] = [4, 2, 4, 3, 3]$$

Having established the polymorphic type List categorically as a **Set**-endofunctor, we consider the collection types Bag and Set. As with List, our goal is show that the two are **Set**-endofunctors. As a first attempt, we stick to the *insert representation* and continue using the polymorphic signature functor associated with List.

Obviously, for a fixed A both τ_A^{Bag} and τ_A^{Set} belong to $\mathbf{Alg}(\mathsf{F}_A)$. However, the two algebras are constrained by additional axioms that capture the semantics of the corresponding type. First, the order of element insertion is not relevant for the constructed values, captured by the so-called *commutativity axiom*

$$cons_A^{\mathsf{T}}\,a'\,(cons_A^{\mathsf{T}}\,a\,as) = cons_A^{\mathsf{T}}\,a\,(cons_A^{\mathsf{T}}\,a'\,as) \tag{Ins-Comm}$$

for all $A \in \mathbf{Set}$ and $\mathsf{T} \in \{\mathsf{Bag}, \mathsf{Set}\}$. Second, for all $A \in \mathbf{Set}$ and $\mathsf{T} = \mathsf{Set}$, inserting an element twice does not affect the value, captured by the so-called *idempotence axiom*.

$$cons_A^{\mathsf{T}}\,a\,(cons_A^{\mathsf{T}}\,a\,as) = cons_A^{\mathsf{T}}\,a\,as \tag{Ins-Idem}$$

The controlled form of confusion introduced by these axioms means that the carriers $\mathsf{Bag}A$ and $\mathsf{Set}A$ are not initial in $\mathbf{Alg}(\mathsf{F}_A)$. This prohibits us from using the Type Former and Type Functor definitions in order to establish Bag and Set as functors. To overcome this limitation, we generalize signature functors to specifications, rendering both $\mathsf{Bag}A$ and $\mathsf{Set}A$ initial in the category of models satisfying the corresponding specification.

Algebraic Specification. Let F be a polynomial signature endofunctor in **Set**, with F-algebras $\alpha : \mathsf{F}A \to A$ factoring into a family of functions $\alpha_i : X_i \to A$. Let E denote a set of equations relating expressions defined solely in terms of α_i and variables universally qualified over their type. Then the pair (F, E) is called an *algebraic specification*.

We can now define algebraic specifications for various collection types by pairing the

partially applied signature functor Ins-Sign with subsets of {Ins-Comm, Ins-Idem}.

$$\begin{aligned}\mathsf{Spec}_A^{\mathsf{List}} &= (\mathsf{F}_A, \varnothing)\\ \mathsf{Spec}_A^{\mathsf{Bag}} &= (\mathsf{F}_A, \{\text{Ins-Comm}\})\\ \mathsf{Spec}_A^{\mathsf{Set}} &= (\mathsf{F}_A, \{\text{Ins-Comm}, \text{Ins-Idem}\})\end{aligned}$$

The containment relation between the equation sets in the above specifications induces the so-called *Boom hierarchy of types* [Bir87].

Now, given a specification $\mathsf{Spec} = (\mathsf{F}, E)$, consider all F-algebras satisfying the equations in E and all F-homomorphisms between them. This subset of $\mathbf{Alg}(\mathsf{F})$ constitutes a subcategory, denoted $\mathbf{Mod}(\mathsf{Spec})$, sometimes also called the *classical semantics* of Spec. The objects in $\mathbf{Mod}(\mathsf{Spec})$ are called *models for* Spec, and morphisms in $\mathbf{Mod}(\mathsf{Spec})$ are called Spec-*homomorphisms*.

To illustrate, consider the F_{Int}-algebras $sum = 0 \triangledown +$ and $min = \infty \triangledown min_2$. Both are contained in $\mathbf{Mod}(\mathsf{Spec}_{Int}^{\mathsf{Bag}})$, because both satisfy Ins-Comm. However, since $+$ is not idempotent, *sum* does not satisfy Ins-Idem and only *min* is a model for $\mathsf{Spec}_{Int}^{\mathsf{Set}}$.

As a final step, the following fact allows for constructing initial objects in $\mathbf{Mod}(\mathsf{Spec})$ from initial objects in the underlying category $\mathbf{Alg}(\mathsf{F})$.

Quotient F-Algebra. Let F be an endofunctor in **Set** such that $\mathbf{Alg}(\mathsf{F})$ has an initial object $\tau : \mathsf{F}T \to T$ and $\mathsf{Spec} = (\mathsf{F}, E)$ be an associated algebraic specification. Relate with $t \sim t'$ elements of T identified (via τ^{-1}) with terms rendered equal by some equation in E. Complete $\sim$ to an equivalence relation $\cong$ by taking its transitive closure. Let $T' = T_{/\cong}$ denote the quotient set of T w.r.t. $\cong$, that is the set of equivalence classes $[t]$ of $t \in T$. Construct the *quotient algebra* $\tau' : \mathsf{F}T' \to T'$ component-wise, defining $\tau'_i : X'_i \to T'$ in terms of its corresponding τ-component $\tau_i : X_i \to T$ using the general scheme $\tau'_i\,[t]\,x = [\tau_i\,t\,x]$. Basically, the scheme adapts all equivalence class parameters $[t] \in T'$ as regular parameters $t \in T$, calls τ_i, and finally maps the result of the τ_i call $r \in T$ to its equivalence class $[r] \in T'$. The quotient algebra is then an isomorphism in **Set**, and consequently τ' is an initial object in $\mathbf{Mod}(\mathsf{Spec})$.

For a proof of this statement, see Theorem 3.7 in [EM85]. For a fixed $A \in \mathbf{Set}$ and $\mathsf{T} \in \{\mathsf{Bag}, \mathsf{Set}\}$ we can use Quotient F-Algebra to define the initial object $\tau_A^{\mathsf{T}} \in \mathbf{Mod}(\mathsf{Spec}_A^{\mathsf{T}})$ in terms of $\tau_A^{\mathsf{List}} \in \mathbf{Alg}(\mathsf{F}_A)$ and the equivalence classes $[\cdot]$ induced by the $\mathsf{Spec}_A^{\mathsf{T}}$ equations.

$$\begin{aligned} &emp_A^{\mathsf{T}} : 1 \to \mathsf{T}A && \text{defined as} && emp_A^{\mathsf{T}}() = [emp_A^{\mathsf{List}}()]\\ &cons_A^{\mathsf{T}} : A \times \mathsf{T}A \to \mathsf{T}A && \text{defined as} && cons_A^{\mathsf{T}}\,[a]\,bs = [cons_A^{\mathsf{List}}\,a\,bs]\end{aligned}$$

Further, properties of catamorphisms transfer from $\mathbf{Alg}(\mathsf{F}_A)$ to $\mathbf{Mod}(\mathsf{Spec}_A^{\mathsf{T}})$. This allows to establish Bag and Set as type functors, reusing Type Former and Type Functor.

Note that the model-theoretic perspective also characterizes situations where a catamorphic definition of a function does not exist. To illustrate one such scenario, consider again the F_{Int}-algebra $sum = 0 \triangledown +$. Specializing Ins-Fold yields a family of catamorphisms $(\!|sum|\!)^{\mathsf{T}}_{Int} : \mathsf{T}Int \to Int$ for $\mathsf{T} \in \{\mathsf{List}, \mathsf{Bag}, \mathsf{Set}\}$ with the following definitions.

$$(\!|sum|\!)^{\mathsf{T}}_{Int}\,(emp^{\mathsf{T}}_{Int}()) = 0$$
$$(\!|sum|\!)^{\mathsf{T}}_{Int}\,(cons^{\mathsf{T}}_{Int}\,a\,as) = a + ((\!|sum|\!)^{\mathsf{T}}_{Int}\,as)$$

Since $sum \notin \mathbf{Mod}(\mathsf{Spec}^{\mathsf{Set}}_{Int})$, the catamorphism $(\!|sum|\!)^{\mathsf{Set}}_{Int}$ does not exist. In other words, in this case the above definition does not constitute a well-defined function, as witnessed by the following derivation of $1 = 2$.

$$\begin{aligned}
& 1 + 0 \\
= \quad & \{\,\text{definition of } (\!|sum|\!)^{\mathsf{Set}}_{Int}\text{, backwards}\,\} \\
& (\!|sum|\!)^{\mathsf{Set}}_{Int}\,(cons^{\mathsf{Set}}_{Int}\,1\,emp^{\mathsf{Set}}_{Int}) \\
= \quad & \{\,\textsc{Ins-Idem}\text{ axiom for }\mathsf{Set}\,\} \\
& (\!|sum|\!)^{\mathsf{Set}}_{Int}\,(cons^{\mathsf{Set}}_{Int}\,1\,(cons^{\mathsf{Set}}_{Int}\,1\,emp^{\mathsf{Set}}_{Int})) \\
= \quad & \{\,\text{definition of } (\!|sum|\!)^{\mathsf{Set}}_{Int}\text{, forwards}\,\} \\
& 1 + 1 + 0
\end{aligned}$$

4.1.5 Collection Types in Union Representation

Using the constructions from Section 4.1.4, we can define an alternative view of the List, Bag, and Set types known as *union representation*. It is based on the signature functor

$$\mathsf{G}(A, B) = 1 + A + B \times B \qquad \text{(Uni-Sign)}$$

and the polymorphic initial G-algebra

$$\forall A \in \mathbf{Set}.\quad \tau^{\mathsf{T}}_A = emp^{\mathsf{T}}_A \triangledown sng^{\mathsf{T}}_A \triangledown uni^{\mathsf{T}}_A : \mathsf{G}_A(\mathsf{T}A) \to \mathsf{T}A \qquad \text{(Uni-Ctor)}$$

where $emp^{\mathsf{T}}_A()$ denotes the empty collection of type $\mathsf{T}A$ (as before), $sng^{\mathsf{T}}_A\,a$ denotes the $\mathsf{T}A$ collection containing a single element $a \in A$, and $uni^{\mathsf{T}}_A\,xs\,ys$ denotes the union of two $\mathsf{T}A$ collections xs and ys. Type constructors are constrained by the following axioms (we omit the superscript T and subscript A for readability)

$$uni\,xs\,emp() = xs \quad \& \quad uni\,emp()\,xs = xs \qquad \text{(Uni-Unit)}$$
$$uni\,xs\,(uni\,ys\,zs) = uni\,(uni\,xs\,ys)\,zs \qquad \text{(Uni-Asso)}$$
$$uni\,xs\,ys = uni\,ys\,xs \qquad \text{(Uni-Comm)}$$

$$uni\ xs\ (uni\ xs\ ys) = uni\ xs\ ys \tag{Uni-Idem}$$

and the associated type functors are induced over the following specifications.

$$\begin{aligned}
\mathsf{Spec}_A^{\mathsf{List}} &= (\mathsf{G}_A, \{\text{Uni-Unit}, \text{Uni-Asso}\}) \\
\mathsf{Spec}_A^{\mathsf{Bag}} &= (\mathsf{G}_A, \{\text{Uni-Unit}, \text{Uni-Asso}, \text{Uni-Comm}\}) \\
\mathsf{Spec}_A^{\mathsf{Set}} &= (\mathsf{G}_A, \{\text{Uni-Unit}, \text{Uni-Asso}, \text{Uni-Comm}, \text{Uni-Idem}\})
\end{aligned}$$

As with Ins-Map, the morphism mapping $\mathsf{T}f = (\!|emp_B^{\mathsf{T}} \triangledown (sng_B^{\mathsf{T}} \circ f) \triangledown uni_B^{\mathsf{T}}|\!)_A^{\mathsf{T}}$ follows from Type Functor and encodes the a structural recursion scheme.

$$\begin{aligned}
\mathsf{T}f\ (emp_A^{\mathsf{T}}()) &= emp_B^{\mathsf{T}}() \\
\mathsf{T}f\ (sng_A^{\mathsf{T}}\ a) &= sng_B^{\mathsf{T}}\ (f a) \\
\mathsf{T}f\ (uni_A^{\mathsf{T}}\ xs\ ys) &= uni_B^{\mathsf{T}}\ (\mathsf{T}f\ xs)\ (\mathsf{T}f\ ys)
\end{aligned} \tag{Uni-Map}$$

Collection types in Spark and Flink do not guarantee element order and allow for duplicates, so they are most accurately modeled by the Bag type constructor. Furthermore, these collections are partitioned across multiple nodes in a shared-nothing cluster, and their value is defined as the disjoint union of all its partitions: $xs = \bigcup\!\!\!\cdot_i^n xs_i$. The union representation supports this view directly, which is why for the purposes of this thesis we prefer Uni-Sign over Ins-Sign. The utility Uni-Sign becomes more evident when we consider the associated structural recursion scheme. Let $\alpha = zero_\alpha \triangledown init_\alpha \triangledown plus_\alpha : \mathsf{G}_A\ B \to B$ be an arbitrary algebra in union representation. Then for $\mathsf{T} \in \{\mathsf{List}, \mathsf{Bag}, \mathsf{Set}\}$ the catamorphism $(\!|\alpha|\!)_A^{\mathsf{T}}$ is recursively defined in terms of the G_A structure as follows.

$$\begin{aligned}
(\!|\alpha|\!)_A^{\mathsf{T}}\ (emp_A^{\mathsf{T}}()) &= zero_\alpha() \\
(\!|\alpha|\!)_A^{\mathsf{T}}\ (sng_A^{\mathsf{T}}\ a) &= init_\alpha\ a \\
(\!|\alpha|\!)_A^{\mathsf{T}}\ (uni_A^{\mathsf{T}}\ as_1\ as_2) &= plus_\alpha\ ((\!|\alpha|\!)_A^{\mathsf{T}}\ as_1)\ ((\!|\alpha|\!)_A^{\mathsf{T}}\ as_2)
\end{aligned} \tag{Uni-Fold}$$

The key insight is that the properties of algebras in union representation ensure a parallel execution scheme regardless of the concrete choice of *zero*, *init*, and *plus*. For a distributed collection $as = \bigcup\!\!\!\cdot_i^n as_i$, a generalized version of the homomorphism property of *uni*

$$(\!|\alpha|\!)_A^{\mathsf{T}} \left(\bigcup\!\!\!\cdot_i^n as_i \right) = (\!|\alpha|\!)_A^{\mathsf{T}} \left(\bigcup\!\!\!\cdot_i^n (\!|\alpha|\!)_A^{\mathsf{T}}\ as_i \right) \tag{Uni-Fold-Dist}$$

implies that we can evaluate $(\!|\alpha|\!)_A^{\mathsf{T}}$ independently on each partition as_i using function shipping, and then fold the Bag of partial results on a single machine in order to compute the final result. In the functional programming community, this idea was highlighted by Steele [Jr.09]. In the Flink and Spark communities, the underlying mathematical principles seem to go largely unnoticed, although their utility has been

recently demonstrated by projects like Summmingbird [BROL14] and MRQL [FLG12]. The relevance of the Uni-Fold recursion scheme is also indicated by the fact that variations of it (under different names), as well as derived operators such as *reduce* and *count* are an integral part in the Flink and Spark APIs.

To conclude, we compare the expressiveness of the two representations. For the rest of this section, let objects, morphisms, and categories associated with the insert representation be color-coded in red, and those associated with the union representation in blue. The fact that we can define the type former $\mathsf{T} \in \{\mathsf{List}, \mathsf{Bag}, \mathsf{Set}\}$ using either Ins-Sign or Uni-Sign implies that the carries of the initial F- and G-algebras in the respective model categories are isomorphic. The isomorphism is established by a pair of catamorphisms $(\!|i2u|\!)_A^{\mathsf{T_F}} \in \mathbf{Mod}(\mathsf{Spec}_A^{\mathsf{T_F}})$ and $(\!|u2i|\!)_A^{\mathsf{T_G}} \in \mathbf{Mod}(\mathsf{Spec}_A^{\mathsf{T_G}})$. The components of the polymorphic algebras $i2u = e \triangledown c$ and $u2i = e \triangledown s \triangledown u$ are defined in terms of constructors in the respective target representation (we omit f_A^{T} from algebra components f for clarity).

$$\begin{aligned} e() &= emp() \\ s\,a &= cons\,a\,emp \\ u\,as_1\,as_2 &= (\!|as_2 \triangledown cons|\!)_A^{\mathsf{T_F}}\,as_1 \end{aligned} \qquad\qquad \begin{aligned} e() &= emp() \\ c\,a\,as &= uni\,(sng\,a)\,as \end{aligned}$$

A natural follow-up question is whether the above correspondence generalizes to non-initial algebras in the F-based model category $\mathbf{Mod}(\mathsf{Spec}_A^{\mathsf{T_F}})$ and the G-based $\mathbf{Mod}(\mathsf{Spec}_A^{\mathsf{T_G}})$. The G-to-F part can be derived from the initial case depicted above – given a G-algebra $(\!|z \triangledown i \triangledown \oplus|\!)_A^{\mathsf{T_G}} \in \mathbf{Mod}(\mathsf{Spec}_A^{\mathsf{T_G}})$, we can construct the corresponding F-algebra $(\!|z \triangledown \otimes|\!)_A^{\mathsf{T_F}} \in \mathbf{Mod}(\mathsf{Spec}_A^{\mathsf{T_F}})$ component-wise as follows.

$$\begin{aligned} z() &= z() \\ a \otimes b &= (i\,a) \oplus b \end{aligned}$$

Similarly, one might think that the same generalization applies in the F-to-G direction.

$$\begin{aligned} z() &= z() \\ i\,a &= a \otimes z \\ b_1 \oplus b_2 &= (\!|b_2 \triangledown p|\!)_A^{\mathsf{T_F}}\,b_1 \end{aligned}$$

Unfortunately, the above construction is not valid. The reason for this is that z does not have to be a unit of $\otimes$. As a counter-example, consider the G-representation derived for the F-algebra $\alpha = 3 \triangledown +$. For $\mathsf{T_F} \in \{\mathsf{List_F}, \mathsf{Bag_F}\}$ the F-catamorphism $(\!|\alpha|\!)_{Int}^{\mathsf{T_F}}$ is the well-defined function adding 3 to the sum of the elements in the input collection. The following examples illustrate this – the result of $(\!|\alpha|\!)$ does not depend on the representation of its argument (we use infix notation $a : as$ instead of $cons\,a\,as$).

$$(\!|\alpha|\!)\,\{\{1,2\}\} = (\!|\alpha|\!)\,(1 : 2 : emp) = 1 + 2 + 3 = 6$$

$$(\!|\alpha|\!)\ \{\{1,2\}\} = (\!|\alpha|\!)\ (2 : 1 : emp) = 2 + 1 + 3 = 6$$

However, the derived G-catamorphism $(\!|\beta|\!)^{\mathsf{T_F}}_{Int}$ is not a well-defined function – the value of $(\!|\alpha|\!)\ as$ depends on the number of emp and sng calls in the as representation. The following examples illustrate this (using the infix notation $as_1 \uplus as_2$ instead of $uni\ as_1\ as_2$).

$$(\!|\alpha|\!)\ \{\{1,2\}\} = (\!|\alpha|\!)\ ((sng\ 1) \uplus (sng\ 2)) = (1 + 3) + (2 + 3) = 9$$
$$(\!|\alpha|\!)\ \{\{1,2\}\} = (\!|\alpha|\!)\ ((sng\ 1) \uplus (sng\ 2) \uplus emp) = (1 + 3) + (2 + 3) + 3 = 12$$

A well-defined F-to-G translation for catamorphisms does not come for free. Tannen and Subrahmanyam give a solution which adds complexity in requiring function types ($\rightarrow$) as an additional type constructor [TS91]. Suciu and Wong [SW95] give another solution which adds complexity in mapping F-catamorphisms that compute in polynomial time to G-catamorphisms that require exponential space. In general, these solutions support a well-known conjecture – there is no "silver bullet" for automatic program parallelization. Conversely, there is a simple way to translate a parallel program (like a catamorphism in $\mathsf{Spec}^{\mathsf{T_G}}_A$) into an equivalent sequential program (a catamorphism in $\mathsf{Spec}^{\mathsf{T_F}}_A$).

4.1.6 Monads and Monad Comprehensions

The type constructors introduced in the Section 4.1.4 and Section 4.1.5 can be used to formalize collection-based data models. In addition to distributed collections managed by dataflow engines such as Flink and Spark, relations managed by Relational Database Management Systems (RDBMSs) can be also understood in terms of bags of database records. As a formalism for the associated processing model, Bag catamorphisms can be used to define primitives for parallel collection processing such as *map* and *reduce*. However, so far we have not seen how catamorphisms relate to declarative languages such as SQL. To make this connection, we show that collection type constructors can be extended to a structure known as *monad with zero*.

A monad with zero allows for declarative syntax known as *monad comprehensions*. In the Bag monad, the syntax and semantics of Bag comprehensions correspond to the syntax and semantics of `Select-From-Where` expressions in SQL. To illustrate this at the syntactic level, compare the SQL expression

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

with the abstract syntax of the corresponding Bag comprehension

$$[\ (x.b, y.d)\ |\ x \leftarrow xs,\ y \leftarrow ys,\ x.a = y.c\ ]^{\mathsf{Bag}} \quad .$$

Formally, a comprehension $[\ e\ |\ qs\ ]^{\mathsf{T}}$ over a collection monad T consists of a *head*

expression e and a *qualifier* sequence qs. A qualifier is either (i) a *generator* $x \leftarrow xs$ binding the elements of $xs : \mathsf{T}A$ to $x : A$, or (ii) a boolean *guard* p (such as $x.a = y.c$) restricting the combinations of generator bindings that contribute to the final result.

As a programming language construct, comprehensions were first adopted by Haskell. Nowadays, comprehension syntax is also natively supported by some GPLs. Python supports List *comprehensions*, written

$$[\ (x.b, y.d)\ \textbf{for}\ x\ \textbf{in}\ xs\ \textbf{for}\ y\ \textbf{in}\ ys\ \textbf{if}\ x.a = y.c\ ]\ .$$

Similarly, Scala supports `for`-*comprehensions*, written

$$\textbf{for}\ \{\ x \leftarrow xs;\ \ y \leftarrow ys;\ \ \textbf{if}\ x.a = y.c\ \}\ \textbf{yield}\ (x.b, y.d)$$

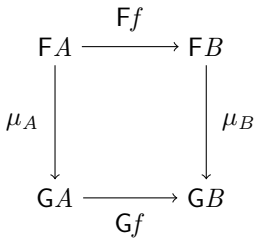
for arbitrary generic types $\mathsf{T}A$ implementing an interface consisting of the functions *map*, *flatMap*, and *withFilter*. We make extensive use of Scala's `for`-comprehensions when designing the source language of our embedded DSL in Chapter 5.

In the remainder of this section, we formally introduce the notions of monad and monad with zero, show how the union-style collection type constructors List, Bag, and Set can be extended to the latter, and define the denotational semantics of monad comprehensions. As a prerequisite, we need to define the notion of a natural transformation.

Natural Transformation. Let $\mathbf{C}$ and $\mathbf{D}$ be two categories and F and G two functors from $\mathbf{C}$ to $\mathbf{D}$. A *natural transformation* $\mu : \mathsf{F} \to \mathsf{G}$ is a family of D-morphisms $\mu_A : \mathsf{F}A \to \mathsf{G}A$ indexed by $A \in \mathbf{C}$, which satisfies the following property for all $\mathbf{C}$-morphisms $f : A \to B$.

$$\mu_B \circ \mathsf{F}f = \mathsf{G}f \circ \mu_A \qquad \text{(Naturality)}$$

The associated commutative diagram is also known as *naturality square*.



If F and G represent polymorphic data containers such as List, Bag, and Set, a natural transformation can be interpreted as container conversion. The Naturality property states that the operation commutes with *map f* applications over the source and target containers – we can either first map over the source container F and then apply the conversion, or convert first and then map over the target container G.

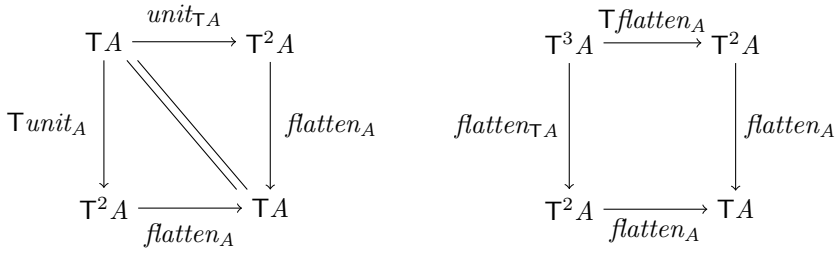
A simple natural transformation was already encountered in Section 4.1.4 – the family of T-constructors $sng_A : A \to \mathsf{T}A$ defines a natural transformation $sng : \mathsf{Id} \to \mathsf{T}$ between Id (which represents the trivial container for an element of type A) and the collection container T. As a second example, consider the List-to-Set conversion defined by the family of List catamorphisms $list2set_A = (\!|emp_A^{\mathsf{Set}} \triangledown sng_A^{\mathsf{Set}} \triangledown uni_A^{\mathsf{Set}}|\!)_A^{\mathsf{List}}$ for all $A \in \mathbf{Set}$. The definition satisfies Naturality – we can either first convert a List to a Set and then apply f to all $a \in \mathsf{Set}A$ or swap the order of these actions with the same net result. Natural transformations and endofunctors are all we need in order to define monads.

Monad. Let $\mathsf{T} : \mathbf{C} \to \mathbf{C}$ be an endofunctor in $\mathbf{C}$. A *monad* is a triple $(\mathsf{T}, unit, flatten)$ consisting of T and two natural transformations – $unit : \mathsf{Id} \to \mathsf{T}$ and $flatten : \mathsf{T}^2 \to \mathsf{T}$ – such that the following properties are satisfied for all $A \in \mathbf{C}$.

$$flatten_A \circ unit_{\mathsf{T}A} = id_{\mathsf{T}A} \quad \& \quad flatten_A \circ \mathsf{T}unit_A = id_{\mathsf{T}A} \qquad \text{(Monad-Unit)}$$

$$flatten_A \circ flatten_{\mathsf{T}A} = flatten_A \circ \mathsf{T}flatten_A \qquad \text{(Monad-Asso)}$$

The above properties are also represented by the following pair of commutative diagrams.



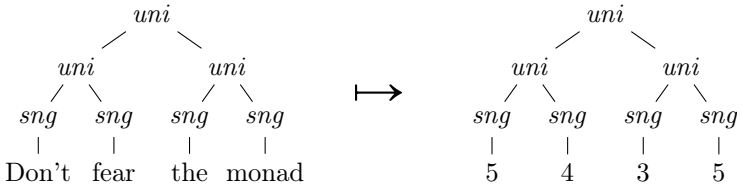
The use of monads in the domain of programming languages semantics can be traced back to Moggi [Mog91]. Conceptually, the type $\mathsf{T}A$ denotes a value of type A attached to some kind of *computational context* (or, in Moggi's parlance, some *notion of computation*) that is modeled by T. Various notions of computation such as optionality, non-determinism, and exceptions can be modeled using a suitable $\mathbf{Set}$-endofunctor T. For the purposes of this thesis, however, we are only interested in situations where T is one of the collection type constructors introduced in Section 4.1.5, and most commonly where $\mathsf{T} = \mathsf{Bag}$. In these cases, the computational context associated with $\mathsf{T}A$ is the context of the constructor application tree where the element values $a \in A$ are inserted in the enclosing collection of type $\mathsf{T}A$. Alternatively, one can say that T represents the computational notion of the state of a collection traversal procedure with cursor pointing at some $a \in A$.

The natural transformation *unit* defines a family of constructors which inject a pure value of type A into a context of type T. If T represents one of the collection type functors encountered in the previous section, we can define *unit* as follows.

$$unit_A^{\mathsf{T}} = sng_A^{\mathsf{T}} \qquad \text{(Uni-Monad-Unit)}$$

The $\mathsf{T}A$-value $unit_A^{\mathsf{T}}\, a$ then denotes the trivial context of a collection containing only a.

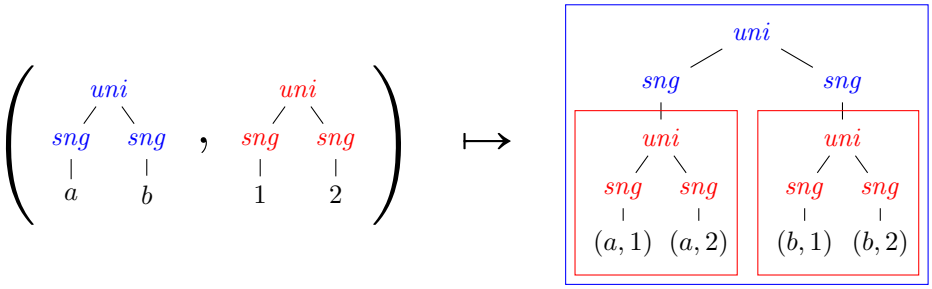
Now consider a situation where we have obtained an instance of a computational context $ta \in \mathsf{T}A$. The morphism component of T allows for transforming the A-values of ta into B-values using some function $f : A \to B$. Crucially, the type of f asserts that f cannot access or modify the enclosing T-context. If T is a collection type functor, then $\mathsf{T}f = map^{\mathsf{T}}\, f$ (as defined in Uni-Map) denotes the function wrapping all ta elements in an f call. The catamorphic interpretation of $map^{\mathsf{T}}\, f\, ta$ is consistent with the monadic perspective – the trees denoted by ta and $map^{\mathsf{T}}\, f\, ta$ have identical shape. To illustrate this, consider the input and output of the application $map^{\mathsf{T}}\, strlen\, \{\!\{ \text{Don't}, \text{fear}, \text{the}, \text{monad} \}\!\}$.



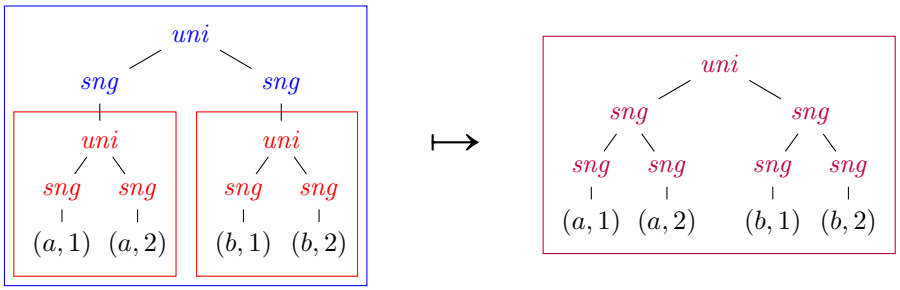
While the input type of f is determined by the ta value type A, its output type can be any object $B \in \mathrm{Ob}_{\mathbf{C}}$. In particular, f might be a value of type $\mathsf{T}B \in \mathbf{C}$, i.e. a value within the same monad. Consequently, $\mathsf{T}f\, ta$ is of type $\mathsf{T}\mathsf{T}B$, shortened as T^2B. In such cases, the natural transformation *flatten* can be used to reduce the structural nesting of the $\mathsf{T}f\, ta$ result. As its name implies, *flatten* transforms a nested context of type T^2A into a flat context of type $\mathsf{T}A$. If T is a collection type functor, T^2 denotes the context of a tree (representing an inner collection) nested within the context of another tree (representing an outer collection). To illustrate this, consider combining the concrete Bag values $xs = \{\!\{a, b\}\!\}$ and $ys = \{\!\{1, 2\}\!\}$ with the following expression.

$$map\,(x \mapsto map\,(y \mapsto (x, y))\, ys)\, xs = \{\!\{\{\!\{(a,1),(a,2)\}\!\}, \{\!\{(b,1),(b,2)\}\!\}\}\!\} \tag{4.1}$$

Expanding Uni-Map at the representation level, the above expression yields an outer collection structurally identical to xs whose values are inner collections identical to ys.



Applying *flatten* on the above result effectively inlines the inner collections in their outer context, resulting in a flat collection of (x, y) values.



The above picture also illustrates the catamorphic nature of $\mathit{flatten}^{\mathsf{T}}$. To flatten a nested T-collection, all we have to do is traverse the (outer) $\mathsf{T}\mathsf{T}A$ tree, inline all (inner) $\mathsf{T}A$ trees passed as arguments of $\mathit{sng}^{\mathsf{T}}_{\mathsf{T}A}$ calls, and change the element type of all $\mathit{emp}^{\mathsf{T}}$ and $\mathit{uni}^{\mathsf{T}}$ constructors from $\mathsf{T}A$ to A. This yields the following definition of $\mathit{flatten}^{\mathsf{T}}$, overloaded for collection type functors $\mathsf{T} = \{\mathsf{List}, \mathsf{Bag}, \mathsf{Set}\}$.

$$\mathit{flatten}^{\mathsf{T}}_{A} = (\!|\mathit{emp}^{\mathsf{T}}_{A} \triangledown \mathit{id}_{\mathsf{T}A} \triangledown \mathit{uni}^{\mathsf{T}}_{A}|\!)^{\mathsf{T}}_{\mathsf{T}A} \qquad \text{(Uni-Monad-Flatten)}$$

We can now interpret the meaning of the Monad-Unit and Monad-Asso properties from the Monad definition.

As we saw above, nesting a $\mathsf{T}g\ tb$ call within the lambda f passed to an outer $\mathsf{T}f\ ta$ call yields a nested monadic type $\mathsf{T}\mathsf{T}A$. Pushing this pattern further, we can imagine nesting of $\mathsf{T}f$ calls of depth n with corresponding results of type $\mathsf{T}^n A$. For simplicity, let us assume that $n = 3$. To transform a value of type $\mathsf{T}\mathsf{T}\mathsf{T}A$ into its flat version of type $\mathsf{T}A$, we need to apply *flatten* twice. The morphism mapping component of the functor T and the polymorphic component A in $\mathit{flatten}_A$ give us two options to achieve this. The first option corresponds to the bracketing $(\mathsf{T}(\mathsf{T}\mathsf{T}))$ – we first flatten the middle and inner T using $\mathsf{T}\mathit{flatten}_A$ (i.e., we map $\mathit{flatten}_A$ over the outer T instance), and then apply $\mathit{flatten}_A$ on the intermediate result. The second option corresponds to the bracketing $((\mathsf{T}\mathsf{T})\mathsf{T})$ – we start with a $\mathit{flatten}_{\mathsf{T}A}$ call which merges the outer and middle T, and then apply $\mathit{flatten}_A$ as in the previous case. The Monad-Asso property states that these two options are equivalent. In other words, *flatten* is *associative* – a nested type $\mathsf{T}^n A$ can be transformed into a flat type $\mathsf{T}A$ by an arbitrary bracketing of *flatten* applications.

Similarly, the polymorphic component A in unit_A and the morphism mapping of T allow for introducing a level of nesting both from the left and from the right of an existing type $\mathsf{T}A$. An application of $\mathsf{T}\mathit{unit}_A$ results in a type-level transformation $\mathsf{T}A \mapsto \mathsf{T}\mathsf{T}A$. Conversely, an application of $\mathit{unit}_{\mathsf{T}A}$ results in a type-level transformation $\mathsf{T}A \mapsto \mathsf{T}\mathsf{T}A$. The Monad-Asso property states if we apply *flatten* after introducing a level of nesting on either side as shown above, we end up with the original $\mathsf{T}A$ value. In other words, $\mathit{flatten}_A$ has $\mathsf{T}\mathit{unit}_A$ as a left unit and $\mathit{unit}_{\mathsf{T}A}$ as a right unit.

As illustrated above, the Monad-Unit and Monad-Asso properties of a monad correspond to the unit and associativity laws of monoidal structures in abstract algebra. This

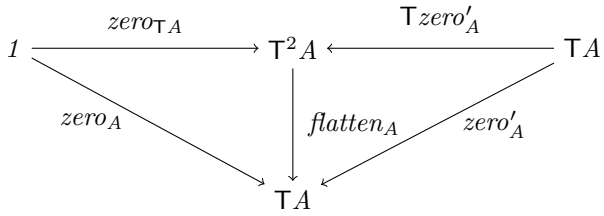
observation motivates the widely cited explanation that "monads are just monoids in the category of endofunctors".

Extending a type functor T to a monad is sufficient in order to define the semantics of comprehension syntax expressions without guards. Adding a zero element to the monad then allows for extending these semantics with support for *guard* qualifiers.

Monad with Zero. Let $zero_A : 1 \to \mathsf{T}A$ be a natural transformation, and $zero'_A : X \to \mathsf{T}A$ denote the composition $zero_A \circ ¡_X$ for arbitrary X. A *monad with zero* is a tuple $(\mathsf{T}, unit, flatten, zero)$ where $(\mathsf{T}, unit, flatten)$ forms a monad and $zero$ satisfies the following properties.

$$flatten_A \circ zero_{\mathsf{T}A} = zero_A \quad \& \quad flatten_A \circ \mathsf{T}zero'_A = zero'_A \qquad \text{(Monad-Zero)}$$

The above equations can be represented by the following pair of commutative triangles.



The Monad-Zero property can be interpreted similarly to Monad-Unit. Given a computational context of type $\mathsf{T}A$, an application of $\mathsf{T}zero'_A$ operates from the inside, replacing arbitrary A-values with an empty $\mathsf{T}A$-context in order to construct a context of type $\mathsf{TT}A$. Conversely, an application of $zero'_{\mathsf{T}A}$ operates from the outside, replacing the entire $\mathsf{T}A$-context with a empty context of type $\mathsf{TT}A$. In both cases, a subsequent *flatten* reduces the resulting nested empty context to its flat form. If T is a collection type functor based on Uni-Sign, the obvious choice is to identify *zero* with *emp*.

$$zero^{\mathsf{T}}_A = emp^{\mathsf{T}}_A \qquad \text{(Uni-Monad-Zero)}$$

Note how Monad-Zero states that *zero* annihilates *flatten*, similar to the way a zero element acts under multiplication in a ring structure. Bringing this similarity further, we can extend a monad with zero with a natural transformation $mplus_A : \mathsf{T}A \times \mathsf{T}A \to \mathsf{T}A$, requiring that (i) *zero* and *mplus* form a monoid and that (ii) *flatten* distributes over *mplus*. As before, if T is a type functor derived from Uni-Sign, using $mplus = uni$ is the obvious choice. The resulting structure $(\mathsf{T}, unit, flatten, zero, mplus)$ is also known as a *ringad*. Ringads were first proposed by Trinder and Wadler [TW89, Tri91] as a formal foundation for comprehensions in database programming languages, and their utility as a formal foundation when reasoning about database optimizations has been recently highlighted by Gibbons [Gib16].

To define the semantics of comprehensions, however, we only need a monad with zero. Here, we use a restricted version of the $\mathcal{MC}$ translation scheme proposed by Grust [GS99], requiring that all qualifiers and the enclosing comprehension are in the same monad T.

$$\begin{aligned}
\mathcal{MC}\,[\,e \mid\,]^{\mathsf{T}} &= \mathit{unit}^{\mathsf{T}}(\mathcal{MC}\ e)\\
\mathcal{MC}\,[\,e \mid q, qs\,]^{\mathsf{T}} &= \mathit{flatten}^{\mathsf{T}}(\mathcal{MC}\,[\,\mathcal{MC}\,[\,e \mid qs\,]^{\mathsf{T}} \mid q\,]^{\mathsf{T}})\\
\mathcal{MC}\,[\,e \mid x \leftarrow xs\,]^{\mathsf{T}} &= \mathit{map}(x \mapsto \mathcal{MC}\ e)(\mathcal{MC}\ xs) \qquad (\mathcal{MC})\\
\mathcal{MC}\,[\,e \mid p\,]^{\mathsf{T}} &= \texttt{if}\ \mathcal{MC}\ p\ \texttt{then}\ \mathit{unit}^{\mathsf{T}}(\mathcal{MC}\ e)\ \texttt{else}\ \mathit{zero}^{\mathsf{T}}\\
\mathcal{MC}\ e &= e
\end{aligned}$$

As an example of the $\mathcal{MC}$ scheme at work, observe how the Bag comprehension

$$[\,(x, y) \mid x \leftarrow xs,\ y \leftarrow ys\,]^{\mathsf{Bag}}$$

translates to the left-hand side of Equation 4.1 followed by $\mathit{flatten}^{\mathsf{Bag}}$.

$$\begin{aligned}
& \mathcal{MC}\,[\,(x, y) \mid x \leftarrow xs,\ y \leftarrow ys\,]^{\mathsf{Bag}}\\
= \quad & \{\ \text{apply}\ \mathcal{MC}\,[\,e \mid q, qs\,]^{\mathsf{T}}\ \}\\
& \mathit{flatten}^{\mathsf{Bag}}(\mathcal{MC}\,[\,\mathcal{MC}\,[\,(x, y) \mid y \leftarrow ys\,]^{\mathsf{Bag}} \mid x \leftarrow xs\,]^{\mathsf{Bag}})\\
= \quad & \{\ \text{apply}\ \mathcal{MC}\,[\,e \mid x \leftarrow xs\,]^{\mathsf{T}}\ \}\\
& \mathit{flatten}^{\mathsf{Bag}}(\mathit{map}\,(x \mapsto \mathcal{MC}\,[\,(x, y) \mid y \leftarrow ys\,]^{\mathsf{Bag}})\,xs)\\
= \quad & \{\ \text{apply}\ \mathcal{MC}\,[\,e \mid x \leftarrow xs\,]^{\mathsf{T}}\ \}\\
& \mathit{flatten}^{\mathsf{Bag}}(\mathit{map}\,(x \mapsto \mathit{map}\,(y \mapsto (x, y))\,ys)\,xs)
\end{aligned}$$

To conclude the section, we prove the following fact. The proof uses the Cata-Map-Fusion law, which is listed in Section 4.1.7.

T-Monads with Zero. A type functor $\mathsf{T} \in \{\mathsf{List}, \mathsf{Bag}, \mathsf{Set}\}$ equipped with $\mathit{unit}^{\mathsf{T}}$, $\mathit{flatten}^{\mathsf{T}}$, and $\mathit{zero}^{\mathsf{T}}$ as defined in Uni-Monad-Unit, Uni-Monad-Flatten, and Uni-Monad-Zero forms a monad with zero.

Proof. We first verify that $\mathit{unit}^{\mathsf{T}}$, $\mathit{flatten}^{\mathsf{T}}$, and $\mathit{zero}^{\mathsf{T}}$ are natural transformations. Naturality follows immediately from the *free theorems* of Wadler [Wad89] and the parametric function types associated with the three functions. For the sake of completeness, we list categorical proofs based on calculational reasoning.

To show that $\mathit{sng}^{\mathsf{T}} : \mathsf{Id} \to \mathsf{T}$ is a natural transformation, verify Naturality as follows.

$$\begin{aligned}
& \mathsf{T}f \circ \mathit{sng}^{\mathsf{T}}_{A}\\
= \quad & \{\ \text{Uni-Map}\ \}
\end{aligned}$$

$$
\begin{array}{ll}
 & sng^{\mathsf{T}}_B \circ f \\
= & \quad \{ \text{ Identity Functor } \} \\
 & sng^{\mathsf{T}}_B \circ \mathsf{Id} f
\end{array}
$$

To show that $emp^{\mathsf{T}} : \mathsf{K}_1 \to \mathsf{T}$ is a natural transformation, verify NATURALITY as follows.

$$
\begin{array}{ll}
 & \mathsf{T}f \circ emp^{\mathsf{T}}_A \\
= & \quad \{ \text{ UNI-MAP } \} \\
 & emp^{\mathsf{T}}_B \\
= & \quad \{ \text{ CATEGORY } \} \\
 & emp^{\mathsf{T}}_B \circ id_1 \\
= & \quad \{ \text{ Constant Functor } \} \\
 & emp^{\mathsf{T}}_B \circ \mathsf{K}_1 f
\end{array}
$$

To show that $flatten^{\mathsf{T}} : \mathsf{TT} \to \mathsf{T}$ is a natural transformation, use the fact that $\mathsf{T}f$ is a $\mathsf{Spec}^{\mathsf{T}}_{\mathsf{T}A}$-homomorphism between the following two $\mathsf{G}_{\mathsf{T}A}$-algebras.

$$
\begin{aligned}
emp^{\mathsf{T}}_A \triangledown id_{\mathsf{T}A} \triangledown uni^{\mathsf{T}}_A &: \mathsf{G}_{\mathsf{T}A}(\mathsf{T}A) \to \mathsf{T}B \\
emp^{\mathsf{T}}_B \triangledown \mathsf{T}f \triangledown uni^{\mathsf{T}}_B &: \mathsf{G}_{\mathsf{T}A}(\mathsf{T}B) \to \mathsf{T}B
\end{aligned}
$$

This is verified as follows.

$$
\begin{array}{ll}
 & \mathsf{T}f \circ (emp^{\mathsf{T}}_A \triangledown id_{\mathsf{T}A} \triangledown uni^{\mathsf{T}}_A) \\
= & \quad \{ \text{ COPRODUCT-FUSION } \} \\
 & (\mathsf{T}f \circ emp^{\mathsf{T}}_A) \triangledown (\mathsf{T}f \circ id_{\mathsf{T}A}) \triangledown (\mathsf{T}f \circ uni^{\mathsf{T}}_A) \\
= & \quad \{ \text{ UNI-MAP } \} \\
 & emp^{\mathsf{T}}_B \triangledown \mathsf{T}f \triangledown (uni^{\mathsf{T}}_B \circ \mathsf{T}f \times \mathsf{T}f) \\
= & \quad \{ \text{ CATEGORY } \} \\
 & (emp^{\mathsf{T}}_B \circ id_1) \triangledown (\mathsf{T}f \circ id_{\mathsf{T}A}) \triangledown (uni^{\mathsf{T}}_B \circ \mathsf{T}f \times \mathsf{T}f) \\
= & \quad \{ \text{ COPRODUCT-FUNCTOR-FUSION } \} \\
 & (emp^{\mathsf{T}}_B \triangledown \mathsf{T}f \triangledown uni^{\mathsf{T}}_B) \circ (id_1 + id_{\mathsf{T}A} + \mathsf{T}f \times \mathsf{T}f) \\
= & \quad \{ \text{ UNI-SIGN } \} \\
 & (emp^{\mathsf{T}}_B \triangledown \mathsf{T}f \triangledown uni^{\mathsf{T}}_B) \circ \mathsf{G}_{\mathsf{T}A}(\mathsf{T}f)
\end{array}
$$

Naturality of $\mathit{flatten}^{\mathsf{T}} : \mathsf{TT} \to \mathsf{T}$ is then established by the following calculation.

$$
\begin{aligned}
& \mathsf{T}f \circ \mathit{flatten}^{\mathsf{T}}_A \\
= & \quad \{ \text{Uni-Monad-Flatten} \} \\
& \mathsf{T}f \circ (\!|\mathit{emp}^{\mathsf{T}}_A \triangledown \mathit{id}_{\mathsf{T}A} \triangledown \mathit{uni}^{\mathsf{T}}_A |\!)^{\mathsf{T}}_{\mathsf{T}A} \\
= & \quad \{ \text{Cata-Fusion (because } \mathsf{T}f \text{ is a } \mathsf{Spec}^{\mathsf{T}}_{\mathsf{T}A}\text{-homomorphism)} \} \\
& (\!|\mathit{emp}^{\mathsf{T}}_B \triangledown \mathsf{T}f \triangledown \mathit{uni}^{\mathsf{T}}_B |\!)^{\mathsf{T}}_{\mathsf{T}A} \\
= & \quad \{ \text{Category, Product Functor} \} \\
& (\!|(\mathit{emp}^{\mathsf{T}}_B \circ \mathit{id}_1) \triangledown (\mathit{id}_{\mathsf{T}B} \circ \mathsf{T}f) \triangledown (\mathit{uni}^{\mathsf{T}}_B \circ \mathit{id}_{\mathsf{T}B} \times \mathit{id}_{\mathsf{T}B}) |\!)^{\mathsf{T}}_{\mathsf{T}A} \\
= & \quad \{ \text{Coproduct-Functor-Fusion} \} \\
& (\!|(\mathit{emp}^{\mathsf{T}}_B \triangledown \mathit{id}_{\mathsf{T}B} \triangledown \mathit{uni}^{\mathsf{T}}_B) \circ (\mathit{id}_1 + \mathsf{T}\mathsf{f} + \mathit{id}_{\mathsf{T}B} \times \mathit{id}_{\mathsf{T}B}) |\!)^{\mathsf{T}}_{\mathsf{T}A} \\
= & \quad \{ \text{Uni-Sign} \} \\
& (\!|(\mathit{emp}^{\mathsf{T}}_B \triangledown \mathit{id}_{\mathsf{T}B} \triangledown \mathit{uni}^{\mathsf{T}}_B) \circ \mathsf{G}(\mathsf{T}f, \mathit{id}_{\mathsf{T}B}) |\!)^{\mathsf{T}}_{\mathsf{T}A} \\
= & \quad \{ \text{Cata-Map-Fusion} \} \\
& (\!|\mathit{emp}^{\mathsf{T}}_B \triangledown \mathit{id}_{\mathsf{T}B} \triangledown \mathit{uni}^{\mathsf{T}}_B |\!)^{\mathsf{T}}_{\mathsf{T}B} \circ \mathsf{T}(\mathsf{T}f) \\
= & \quad \{ \text{Uni-Monad-Flatten, functor composition} \} \\
& \mathit{flatten}^{\mathsf{T}}_B \circ \mathsf{TT}f
\end{aligned}
$$

The proof that $(\mathsf{T}, \mathit{unit}^{\mathsf{T}}, \mathit{flatten}^{\mathsf{T}})$ constitutes a Monad is completed in two steps. First, we show that Uni-Monad-Unit satisfies the two Monad-Unit equations.

Right unit:

$$
\begin{aligned}
& \mathit{flatten}^{\mathsf{T}}_A \circ \mathsf{T}\mathit{unit}^{\mathsf{T}}_A \\
= & \quad \{ \text{Uni-Monad-Flatten, Uni-Monad-Unit} \} \\
& (\!|\mathit{emp}^{\mathsf{T}}_A \triangledown \mathit{id}_{\mathsf{T}A} \triangledown \mathit{uni}^{\mathsf{T}}_A |\!)^{\mathsf{T}}_{\mathsf{T}A} \circ \mathsf{T}\mathit{sng}^{\mathsf{T}}_A \\
= & \quad \{ \text{Cata-Map-Fusion} \} \\
& (\!|(\mathit{emp}^{\mathsf{T}}_A \triangledown \mathit{id}_{\mathsf{T}A} \triangledown \mathit{uni}^{\mathsf{T}}_A) \circ \mathsf{G}(\mathit{sng}^{\mathsf{T}}_A, \mathit{id}_{\mathsf{T}A}) |\!)^{\mathsf{T}}_A \\
= & \quad \{ \text{Uni-Sign} \} \\
& (\!|(\mathit{emp}^{\mathsf{T}}_A \triangledown \mathit{id}_{\mathsf{T}A} \triangledown \mathit{uni}^{\mathsf{T}}_A) \circ (\mathit{id}_1 + \mathit{sng}^{\mathsf{T}}_A + \mathit{id}_{\mathsf{T}A} \times \mathit{id}_{\mathsf{T}A}) |\!)^{\mathsf{T}}_A \\
= & \quad \{ \text{Coproduct-Functor-Fusion} \} \\
& (\!|\mathit{emp}^{\mathsf{T}}_A \triangledown \mathit{sng}^{\mathsf{T}}_A \triangledown \mathit{uni}^{\mathsf{T}}_A |\!)^{\mathsf{T}}_A \\
= & \quad \{ \text{Cata-Reflect} \} \\
& \mathit{id}_{\mathsf{T}A}
\end{aligned}
$$

Left unit:

$$
\begin{aligned}
& \mathit{flatten}^{\mathsf{T}}_{A} \circ \mathit{unit}^{\mathsf{T}}_{\mathsf{T}A} \\
= \quad & \{\ \textsc{Uni-Monad-Flatten},\ \textsc{Uni-Monad-Unit}\ \} \\
& (\!|\mathit{emp}^{\mathsf{T}}_{A} \triangledown \mathit{id}_{\mathsf{T}A} \triangledown \mathit{uni}^{\mathsf{T}}_{A}|\!)^{\mathsf{T}}_{\mathsf{T}A} \circ \mathit{sng}^{\mathsf{T}}_{\mathsf{T}A} \\
= \quad & \{\ \textsc{Uni-Fold}\ \} \\
& \mathit{id}_{\mathsf{T}A}
\end{aligned}
$$

Second, show that Uni-Monad-Unit and Uni-Monad-Flatten satisfy Monad-Asso. Again, the proof relies on the auxiliary fact that $\mathit{flatten}^{\mathsf{T}}_{A}$ is a $\mathsf{Spec}^{\mathsf{T}}_{\mathsf{TT}A}$-homomorphism between the following two $\mathsf{G}_{\mathsf{TT}A}$-algebras

$$
\begin{aligned}
\mathit{emp}^{\mathsf{T}}_{\mathsf{T}A} \triangledown \mathit{id}_{\mathsf{TT}A} \triangledown \mathit{uni}^{\mathsf{T}}_{\mathsf{T}A} &: \mathsf{G}_{\mathsf{TT}A}(\mathsf{TT}A) \to \mathsf{T}A \\
\mathit{emp}^{\mathsf{T}}_{A} \triangledown \mathit{flatten}^{\mathsf{T}}_{A} \triangledown \mathit{uni}^{\mathsf{T}}_{A} &: \mathsf{G}_{\mathsf{TT}A}(\mathsf{T}A) \to \mathsf{T}A
\end{aligned}
$$

which is verified as follows.

$$
\begin{aligned}
& (\mathit{emp}^{\mathsf{T}}_{A} \triangledown \mathit{flatten}^{\mathsf{T}}_{A} \triangledown \mathit{uni}^{\mathsf{T}}_{A}) \circ \mathsf{G}_{\mathsf{TT}A}\, \mathit{flatten}^{\mathsf{T}}_{A} \\
= \quad & \{\ \textsc{Uni-Sign}\ \} \\
& (\mathit{emp}^{\mathsf{T}}_{A} \triangledown \mathit{flatten}^{\mathsf{T}}_{A} \triangledown \mathit{uni}^{\mathsf{T}}_{A}) \circ (\mathit{id}_{1} + \mathit{id}_{\mathsf{TT}A} + \mathit{flatten}^{\mathsf{T}}_{A} \times \mathit{flatten}^{\mathsf{T}}_{A}) \\
= \quad & \{\ \textsc{Coproduct-Functor-Fusion}\ \} \\
& (\mathit{emp}^{\mathsf{T}}_{A} \triangledown \mathit{flatten}^{\mathsf{T}}_{A} \triangledown (\mathit{uni}^{\mathsf{T}}_{A} \circ \mathit{flatten}^{\mathsf{T}}_{A} \times \mathit{flatten}^{\mathsf{T}}_{A})) \\
= \quad & \{\ \textsc{Uni-Monad-Zero},\ \text{free theorem for } \mathit{uni}^{\mathsf{T}}\ \} \\
& ((\mathit{flatten}^{\mathsf{T}}_{A} \circ \mathit{emp}^{\mathsf{T}}_{\mathsf{T}A}) \triangledown (\mathit{flatten}^{\mathsf{T}}_{A} \circ \mathit{id}_{\mathsf{TT}A}) \triangledown (\mathit{flatten}^{\mathsf{T}}_{A} \circ \mathit{uni}^{\mathsf{T}}_{\mathsf{T}A})) \\
= \quad & \{\ \textsc{Coproduct-Fusion}\ \} \\
& \mathit{flatten}^{\mathsf{T}}_{A} \circ (\mathit{emp}^{\mathsf{T}}_{\mathsf{T}A} \triangledown \mathit{id}_{\mathsf{TT}A} \triangledown \mathit{uni}^{\mathsf{T}}_{\mathsf{T}A})
\end{aligned}
$$

Monad-Asso is then verified by the following calculation.

$$
\begin{aligned}
& \mathit{flatten}^{\mathsf{T}}_{A} \circ \mathit{flatten}^{\mathsf{T}}_{\mathsf{T}A} \\
= \quad & \{\ \textsc{Uni-Monad-Flatten}\ \} \\
& \mathit{flatten}^{\mathsf{T}}_{A} \circ (\!|\mathit{emp}^{\mathsf{T}}_{\mathsf{T}A} \triangledown \mathit{id}_{\mathsf{TT}A} \triangledown \mathit{uni}^{\mathsf{T}}_{\mathsf{T}A}|\!)^{\mathsf{T}}_{\mathsf{TT}A} \\
= \quad & \{\ \textsc{Cata-Fusion}\ (\text{because } \mathit{flatten}^{\mathsf{T}}_{A} \text{ is a } \mathsf{Spec}^{\mathsf{T}}_{\mathsf{TT}A}\text{-homomorphism})\ \} \\
& (\!|\mathit{emp}^{\mathsf{T}}_{A} \triangledown \mathit{flatten}^{\mathsf{T}}_{A} \triangledown \mathit{uni}^{\mathsf{T}}_{A}|\!)^{\mathsf{T}}_{\mathsf{TT}A} \\
= \quad & \{\ \textsc{Coproduct-Functor-Fusion}\ \} \\
& (\!|(\mathit{emp}^{\mathsf{T}}_{A} \triangledown \mathit{id}_{\mathsf{T}A} \triangledown \mathit{uni}^{\mathsf{T}}_{A}) \circ (\mathit{id}_{1} + \mathit{flatten}^{\mathsf{T}}_{A} + \mathit{id}_{\mathsf{T}A} \times \mathit{id}_{\mathsf{T}A})|\!)^{\mathsf{T}}_{\mathsf{TT}A} \\
= \quad & \{\ \textsc{Uni-Sign}\ \}
\end{aligned}
$$

$$
\begin{aligned}
& (\!|(emp^{\mathsf{T}}_A \triangledown id_{\mathsf{T}A} \triangledown uni^{\mathsf{T}}_A) \circ \mathsf{G}(flatten^{\mathsf{T}}_A, id_{\mathsf{T}A})|\!)^{\mathsf{T}}_{\mathsf{T}\mathsf{T}A} \\
= \quad & \{ \text{Cata-Map-Fusion} \} \\
& (\!|emp^{\mathsf{T}}_A \triangledown id_{\mathsf{T}A} \triangledown uni^{\mathsf{T}}_A|\!)^{\mathsf{T}}_{\mathsf{T}A} \circ \mathsf{T} flatten^{\mathsf{T}}_A \\
= \quad & \{ \text{Uni-Monad-Flatten} \} \\
& flatten^{\mathsf{T}}_A \circ \mathsf{T} flatten^{\mathsf{T}}_A
\end{aligned}
$$

Finally, to show that $(\mathsf{T}, unit^{\mathsf{T}}, flatten^{\mathsf{T}}, zero^{\mathsf{T}})$ constitutes a Monad with Zero, we show that Uni-Monad-Zero satisfies the two Monad-Zero equations.

Outer:

$$
\begin{aligned}
& flatten^{\mathsf{T}}_A \circ zero^{\mathsf{T}}_{\mathsf{T}A} \\
= \quad & \{ \text{Uni-Monad-Flatten}, \text{Uni-Monad-Zero} \} \\
& (\!|emp^{\mathsf{T}}_A \triangledown id_{\mathsf{T}A} \triangledown uni^{\mathsf{T}}_A|\!)^{\mathsf{T}}_{\mathsf{T}A} \circ emp^{\mathsf{T}}_{\mathsf{T}A} \\
= \quad & \{ \text{Uni-Fold}, \text{Uni-Monad-Zero} \} \\
& zero^{\mathsf{T}}_A
\end{aligned}
$$

Inner:

$$
\begin{aligned}
& flatten^{\mathsf{T}}_A \circ \mathsf{T}(zero^{\mathsf{T}}_A \circ ¡_A) \\
= \quad & \{ \text{Uni-Monad-Flatten}, \text{Uni-Monad-Zero} \} \\
& (\!|emp^{\mathsf{T}}_A \triangledown id_{\mathsf{T}A} \triangledown uni^{\mathsf{T}}_A|\!)^{\mathsf{T}}_{\mathsf{T}A} \circ \mathsf{T}(emp^{\mathsf{T}}_A \circ ¡_A) \\
= \quad & \{ \text{Cata-Map-Fusion}, \text{Uni-Sign} \} \\
& (\!|(emp^{\mathsf{T}}_A \triangledown id_{\mathsf{T}A} \triangledown uni^{\mathsf{T}}_A) \circ (id_1 + (emp^{\mathsf{T}}_A \circ ¡_A) + id_{\mathsf{T}A} \times id_{\mathsf{T}A})|\!)^{\mathsf{T}}_A \\
= \quad & \{ \text{Coproduct-Functor-Fusion} \} \\
& (\!|emp^{\mathsf{T}}_A \triangledown (emp^{\mathsf{T}}_A \circ ¡_A) \triangledown uni^{\mathsf{T}}_A|\!)^{\mathsf{T}}_A \\
= \quad & \{ \text{Uni-Unit applied in step } (iii) \text{ of the catamorphism} \} \\
& emp^{\mathsf{T}}_A \circ ¡_A \\
= \quad & \{ \text{Uni-Monad-Zero} \} \\
& zero^{\mathsf{T}}_A \circ ¡_A
\end{aligned}
$$

□

4.1.7 Fusion

A number of useful laws can be established from the properties of the collection type functors T and the model categories $\mathsf{Spec}_A^{\mathsf{T}}$. Three of those are of special interest for us, as they explain the optimizing program transformations from Example 2.3. The first law fuses a $\mathsf{Spec}_B^{\mathsf{T}}$-catamorphism $(\!|\beta|\!)_B^{\mathsf{T}}$ with a preceding functor application $\mathsf{T}f : \mathsf{T}A \to \mathsf{T}B$.

$$(\!|\beta|\!)_B^{\mathsf{T}} \circ \mathsf{T}f = (\!|\beta \circ \mathsf{G}(f, id_{\mathsf{T}B})|\!)_A^{\mathsf{T}} \qquad \text{(Cata-Map-Fusion)}$$

To understand how Cata-Map-Fusion relates to the code snippets listed for Example 2.3, observe that *reduce* can be defined as the following catamorphism, where $error : 1 \to 0$ is a function indicating a runtime error.

$$reduce_A^{\mathsf{T}}\, p = (\!|error \triangledown id_A \triangledown p|\!)_A^{\mathsf{T}}$$

Substituting the above definition in Cata-Map-Fusion yields the following equation.

$$reduce_B^{\mathsf{T}}\, p \circ map^{\mathsf{T}} f = (\!|error \triangledown f \triangledown p|\!)_B^{\mathsf{T}}$$

A chain of *map* and *reduce* applications thereby can be fused into a single catamorphism – i.e., they can be evaluated with a single pass over the data. This fact is well known in the area of dataflow engines – systems like Flink and Spark automatically execute chains of *map* and *reduce* operators in a pipelined manner.

The second law fuses a pair of $\mathsf{Spec}_A^{\mathsf{T}}$-catamorphisms with result types B and C into a single $\mathsf{Spec}_A^{\mathsf{T}}$-catamorphism with result type $B \times C$.

$$(\!|\beta|\!)_A^{\mathsf{T}} \times (\!|\gamma|\!)_A^{\mathsf{T}} = (\!|(\beta \circ \mathsf{G}_A\, \pi_B) \times (\gamma \circ \mathsf{G}_A\, \pi_C)|\!)_A^{\mathsf{T}} \qquad \text{(Banana-Split)}$$

Together with Cata-Map-Fusion, the law explains the optimizing program transformation applied to the "movies by Hitchcock and Allen" code in Example 2.3. We first apply Cata-Map-Fusion, fusing the two *map* applications (which return either 1 or 0 depending on the director's name) with the subsequent *reduce* applications (which sum the resulting bag of numbers). Then, we apply Banana-Split and obtain a single catamorphism which accumulates movies by Alfred Hitchcock and Woody Allen simultaneously as a pair of counts. Finally, we apply Cata-Map-Fusion in the reverse direction, resulting in a version of the fused catamorphism expressed as a $reduce^{\mathsf{T}} g \circ map^{\mathsf{T}} f$ dataflow. In Section 7.2.1, we describe how these optimizing rewrites can be performed as part of the *Emma* compiler pipeline.

The third law makes use of the following monadic definition of *groupBy*

$$groupBy\, k\, xs = [\ (k_x, [\ x \mid x \leftarrow xs;\ k\, x = k_x\ ]^{\mathsf{T}}) \mid k_x \leftarrow distinct\, [\ k\, x \mid x \leftarrow xs\ ]^{\mathsf{T}}\ ]^{\mathsf{T}}$$

and fuses a $groupBy\, k\, xs$ with a catamorphism applied to the values of each group.

$$\begin{aligned} & \mathsf{T}(id_K \times (\!|z \triangledown i \triangledown p|\!)^{\mathsf{T}}_A)\,(groupBy\, k\, xs) \\ & = \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(Fold-Group-Fusion)} \\ & [\ (k_x, (\!|z \triangledown \bar{i} \triangledown p|\!)^{\mathsf{T}}_A\, xs) \mid k_x \leftarrow distinct\, [\ k\, x \mid x \leftarrow xs\ ]^{\mathsf{T}}\ ]^{\mathsf{T}} \end{aligned}$$

The function $\bar{i}$ used in the above equation is defined as follows.

$$\bar{i}\, x = \begin{cases} i\, x & \text{if } k\, x = k_x \\ z & \text{otherwise} \end{cases}$$

The insights of Fold-Group-Fusion and Uni-Fold-Dist explain why Spark dataflows should be specified in the shape given by (4.2) and not in the more intuitive shape given by (4.3), as illustrated by he "movies per decade" code snippet from Example 2.3.

$$reduceByKey\, p \circ mapValues\, i \circ keyBy\, k \tag{4.2}$$

$$mapValues\, (vs \Rightarrow (\!|error \triangledown i \triangledown p|\!)\, vs) \circ groupBy\, k \tag{4.3}$$

The (4.2) shape facilitates an execution strategy where the catamorphism $(\!|z \triangledown \bar{i} \triangledown p|\!)^{\mathsf{T}}_A$ is applied twice in parallel – once on each partition xs_i, and once more after partial results are repartitioned based on their k_x value. This strategy is more efficient because it reduces the amount of shuffled data. On the other side, as we already indicated in Chapter 3, dataflows shaped like (4.3) do not permit the same execution strategy because the $vs \Rightarrow (\!|error \triangledown i \triangledown p|\!)\, vs$ function cannot be inspected or modified using the embedding methodology adopted by Spark. An analogous argument holds for Flink. In Section 7.2.2, we propose an automatic optimization which builds on Fold-Group-Fusion and translates *Emma* dataflows with (4.3) shape into dataflows with (4.2) shape.

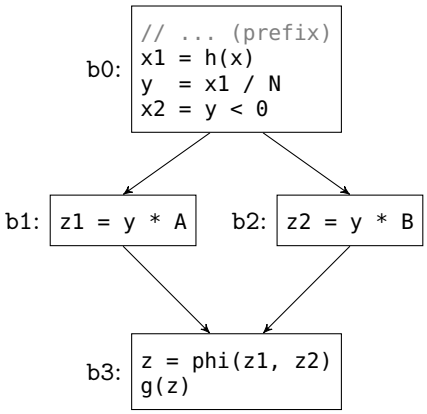
4.2 Static Single Assignment Form

Language compilers typically perform a number of program optimizations. These are usually 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 form and its functional encoding – ANF – have been successfully used in a number of programming language compilers. As the IR of the DSL proposed in this thesis depends on ANF, this section introduces the main ideas behind SSA and ANF based on a simple example (Figure 4.1). For a more thorough primer of these concepts, we refer the reader to the overview paper by Appel [App98].

The source code formulation of the example program (Figure 4.1a) offers various degrees of syntactic freedom. For instance, we could have inlined `y` in its call sites or defined

```
// ... (prefix)
val y = h(x) / N
var z = 0
if (y < 0)
  y * A
else
  y * B
g(z)
```

(a) Source code



(b) SSA graph

```
// ... (prefix)
val x1 = h(x)
var y  = x1 / N
val x2 = y < 0
def k1() = {
  val z1 = y * A
  k3(z1) }
def k2() = {
  val z2 = y * B
  k3(z2) }
def k3(z: Int) = {
  g(z) }
if (x2) k1() else k2()
```

(c) ANF code

Figure 4.1: Example program in source, SSA, and ANF forms.

`z` as a variable assigned in the two branches. Therefore, it is hard to define program analysis on top of a source-code-like syntax tree – we have to accommodate for all forms of variability and their (possibly non-orthogonal) interactions. In contrast, the derived SSA graph (Figure 4.1b) offers a normalized representation of the source code 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 sub-expression `f(x)` is assigned to a fresh variable `x1` and referenced in the division application bound to `y`. Values with control-flow dependent data dependencies 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 `b4` block at runtime.

The SSA graph can be also represented as a functional program in ANF (Figure 4.1c). In this representation, control-flow blocks are encoded as *continuation functions* such as `k1` – `k3`, and control-flow edges are encoded as *continuation calls* such as `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`, which corresponds to the `z` = `phi(z1, z2)` definition in Figure 4.1b.

5 Source Language

To address the problems with state-of-the-art parallel dataflow eDSLs outlined in Chapter 2 we propose *Emma* – a quotation-based DSL embedded in Scala. Section 5.1 outlines linguistic features 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 our quotation-based compiler. Finally, Section 5.3 presents and illustrates the programming abstractions that form the core *Emma* API.

5.1 Linguistic Features and Restrictions

As outlined in Chapter 3, we claim that problems with state-of-the-art eDSLs for parallel collection processing are a consequence of the adopted type-based embedding strategy. The difficulties stem from the fact that program structure critical for optimization is either not represented or treated as a black box in the IR lifted by these eDSLs. To tackle these problems, we analyzed a wide range of algorithms implemented in the Spark `RDD` and Flink `DataSet` eDSLs and identified a set of host language features needed to express these algorithms in a direct and concise way. The ability to freely compose these features at the Scala (source) level and reflect them at the IR level is crucial for any eDSL that wants to attain maximal linguistic reuse without sacrificing optimization potential. The features are:

(**F1**) control-flow primitives such as `if-else`, `while`, and `do-while`;
(**F2**) `var` and `val` definitions as well as `var` assignments;
(**F3**) `lambda` function definitions;
(**F4**) `def` method calls and `new` object instantiations;
(**F5**) statement blocks.

In addition to those, the following Scala features are either defined as syntactic sugar

that *desugars* in terms of (F1-F5) in the Scala ASTs, or they can be eliminated with a simple transformation:

(**F6**) `for`-comprehensions – those are represented as chains of nested `flatMap`, `withFilter`, and `map` calls using a DESUGAR scheme similar to the $\mathcal{MC}$ transformation from in Section 4.1;
(**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 represented as `foreach` calls in the Scala AST and can be subsequently rewritten as `while` loops.

Some restrictions are also made in order to simplify the definition and development of the compiler frontend and the optimizing program transformations presented in the rest of this thesis.

(**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;
(**R7**) `var` assignments outside of their defining scope (i.e., inside a lambda).

We proceed by formalizing a user-facing language called *Emma Source*. The abstract syntax of *Emma Source* models a subset of Scala covering (F1-F5) and therefore can be easily derived from the ASTs of quoted Scala code snippets.

5.2 Abstract Syntax

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

We denote metalanguage expressions in *italic* and object language expressions in a

$t :=$		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*)	$x : T$	param. def
	this	`this` ref			
	module	module ref			
	x	binding ref			

Figure 5.1: Abstract syntax of *Emma Source*.

`teletype` font family. The following example illustrates the difference.

(metalanguage) *xs.take*(*10*) $\Leftrightarrow$ `xs.take(10)` (object language)

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 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 specified in Figure 5.1. The language consists of two mutually recursive definitions – *terms*, which always return a value, and *statements*, which modify the computation state. Some of the more critical aspects of the syntax are discussed below.

First, we assume that *Source* expressions are (*i*) *typed* – that is, every term is related to a unique type $t : T$, and (*ii*) *named* – that is, every definition is related to a unique symbol with an enclosing scope. Both assumptions are already met when using compile-time reflection (i.e, Scala macros), and can be enforced at runtime with an extra *typeCheck*(*expr*) call. This implies that m in method call, X in `new` instance, and x in binding refs denote unique symbols rather than ambiguous names.

Second, the language offers mechanisms for (*i*) abstraction – via lambda terms, and for (*ii*) control-flow – via conditional terms and loop constructs. Crucially, the abstract syntax ensures (*ii*) is stratified with respect to (*i*). Were recursive functions (`def` definitions in Scala) included in *Source*, this assumption would have been violated. This restriction is simplifies the definition of a decision procedure for the concept of *binding context* in Section 6.4.

5.3 Programming Abstractions

The core programming abstraction is a trait `Bag A` which represents a distributed collection with elements of type `A` and a matching `BagCompanion` trait defining various `Bag` constructors. The API is listed in Figure 5.2. To illustrate and outline key differences between the `Bag` and `RDD`/`DataSet` APIs, in the remainder of this section we re-cast some examples from Section 2.3.

5.3.1 Sources and Sinks

The *data source* operators in the `BagCompanion` trait define various `Bag` constructors. For each *source* there is a corresponding *sink* which operates 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
```

5.3.2 Select-From-Where-like Syntax

The operators in the right column in Figure 5.2 define a Scala-native interface for parallel collection processing similar to SQL. Binary operators like `join` and `cross` are omitted from the API. Instead, the `Bag` type implements the monad operations discussed in Section 4.1. This allows for declarative `Select-From-Where`-like expressions using Scala's `for`-comprehension syntax. The joins from Example 2.2 can be expressed as follows.

```
// join movies, credits, and people and build intermediate (m, c, p) triples
val ys = for {
  m <- movies
  c <- credits
  p <- people
  if m.id == c.movieID
  if p.id == c.personID
} yield (m, c, p)
// pattern-match (m, c, p) triples and project final result
for {
 (m, c, p) <- ys
} yield (m.title, p.name)
```

Data sinks (in Bag$A$)

`fetch() : SeqA`

`as[DColl[_]] : DCollA`

`writeParquet(path : String, ...) : Unit`

`writeCSV(path : String, ...) : Unit`

Data sources (in BagCompanion)

`empty[A] : BagA`

`apply[A](values : SeqA) : BagA`

`from[DColl[_], A](coll : DCollA) : BagA`

`readParquet[A](path : String, ...) : BagA`

`readCSV[A](path : String, ...) : BagA`

SQL-like (in Bag$A$)

`map[B](f : A ⇒ B) : BagB`

`flatMap[B](f : A ⇒ BagB : BagB`

`withFilter(p : A ⇒ Boolean) : BagA`

`union(that : BagA) : BagA`

`groupBy[K](k : A ⇒ K) : BagGroup[K, BagA]`

`distinct : BagA`

Folds (in Bag$A$)

`fold[B](alg : Alg[A, B]) : B`

`size : Long = fold(0L)(_ ⇒ 1L, _ + _)`

`nonEmpty, min, max, ...`

Figure 5.2: Bag$A$ and BagCompanion API.

Moreover, maintaining comprehension syntax at the IR level allows us to employ query compilation techniques such as projection- and filter-pushdown (see Section 7.1).

5.3.3 Aggregation and Grouping

Aggregating the values of a `Bag` is based on a single primitive – `fold` – which represents structural recursion over Union-style bags. The method accepts a Union-algebra instance that encapsulates substitution functions for the three basic Union-style bag constructors. The algebra trait `Alg` and an example `Size` algebra 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: Long = 0
  val init: Any => Long = const(1)
  val plus: (Long, Long) => Long = _ + _
}
```

Various 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
```

Per-group aggregations are defined in a straight-forward way using `groupBy` and `for`-comprehensions. Example 2.3 can be written as follows.

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

```
} yield (d, ms.size)
```

Rewriting this definition in terms of primitives such as `reduceByKey` is enabled by (*i*) the insight that structural recursion (i.e, `fold`s) over UNION-style collections models data-parallel computation, and (*ii*) the ability to represent nested `Bag` computations in the IR. Details are discussed in Section 7.2.

5.3.4 Caching and Native Iterations

The API does not require explicit caching. `Bag` terms referenced more than once or inside a loop are implicitly cached (Section 7.3). For example, in

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

both `S` and the inner `w` will be implicitly cached. In addition, we propose API extensions and transformations that rewrite loop structures to Flink's `iterate` operator whenever possible (Section 7.4).

5.3.5 API Implementations

The `Bag` and `BagCompanion` traits are implemented once per backend. At the moment, we implement `FlinkBag` (backed by a Flink `DataSet`) and a `SparkBag` (backed by a Spark `Dataset`). The backend implementation is introduced transparently as part of the compilation pipeline as sketched in Section 6.5. This design allows for introducing other backends in the future. A `ScalaBag` (backed by a Scala `Seq`) is used per default – the `Bag` object just delegates to the `ScalaBag` companion. Unquoted *Emma* code snippets therefore can be executed and debugged as regular Scala programs. Consequently, programmers can first focus on writing correct code without thinking about distributed execution, and quote the code in order to parallelize it later.

6 Core Language

In line with the envisioned optimizations, we propose a functional intermediate language called *Emma Core*. To simplify program analysis, we build on the *A-normal form (ANF)* of Flanagan et al. [FSDF93] presented in Section 4.2. To that end, in Section 6.1 we define *Emma Core*$_{ANF}$ and present a translation scheme from *Emma Source* to *Emma Core*$_{ANF}$. To accommodate for SQL-like program rewrites such as join-order enumeration, in Section 6.2 we add first-class support for monad comprehensions, extending *Emma Core*$_{ANF}$ to *Emma Core*, and in Section 6.3 we sketch a comprehension normalization scheme. Section 6.4 introduces the notion of *binding context*. Finally, Section 6.5 gives an overview of the *Emma* compiler pipeline.

6.1 Administrative Normal Form

The abstract syntax of the *Emma Core*$_{ANF}$ language is specified in Figure 6.1. Below, we outline the main differences between the terms and statements in *Emma Core*$_{ANF}$ and *Emma Source*.

The sub-language of atomic terms (denoted by a) is shared between the two languages. Imperative statement blocks are replaced by functional *let* blocks. Terms that may appear on the right-hand side of `val` definitions are restricted from t to b, ensuring that all sub-terms (except lambda) are atomic. Loops are replaced by continuation functions in the so-called *direct-style*, and `var` definitions and assignments are replaced by continuation parameters. Continuation definitions appear after the *vdefs* sequence in `let` blocks and are called only at the return position c. As noted by Appel [App98] and illustrated in Section 4.2, the resulting functional representation corresponds to the SSA form commonly used in modern compilers. In particular:

- Value assignments are static – every value x is associated with a unique binding definition.

$a :=$	… (as in Figure 5.1)	atomic	$let :=$	$\{\ vdefs;\ kdefs;\ c\ \}$	*let* block
$b :=$		binding			
	$a.m[T_j](as)_i$	method call	$stat :=$		statement
	`new` $X[T_j](as)_i$	`new` instance	$(kdef)$	`def` $k(pdefs) = let$	cont. def
	$pdefs \Rightarrow let$	lambda		$bdef$	binding def
	$a.module$	mod. access	$bdef :=$		binding 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			

Figure 6.1: Abstract syntax of *Emma* $Core_{ANF}$.

- Control-flow blocks are in 1:1 correspondence with `let` blocks. Every block is uniquely identified by its nearest enclosing lambda or continuation definition. If this does not exist, then the block is the (unique) root of the control-flow graph.
- Control-flow edges are in 1:1 correspondence with continuation calls. Calling a continuation k_2 from the let-block enclosed in k_1 implies a control-flow edge from k_1 to k_2.
- SSA ϕ nodes are in 1:1 correspondence with arguments bound to continuation parameters. A continuation `def k(x : X, y : Y) = …` called twice with `k(v, w)` and `k(v′, w′)` implies two ϕ calls $\mathtt{x} = \phi(\mathtt{v}, \mathtt{v}')$ and $\mathtt{y} = \phi(\mathtt{w}, \mathtt{w}')$.
- The dominance tree associated with the control-flow graph is in 1:1 correspondence with the hierarchy induced by nested continuation definitions.

The translation from *Source* to $Core_{ANF}$ is defined as the composition of two distinct transformations. The ANF transformation (Figure 6.2) destructs compound t terms as statement blocks with restricted structure. Each sub-term becomes a named b-term in a `val` definition. The return expression of the resulting block is always atomic. The ANF-VAR and ANF-ASGN rules ensure that terms appearing on the right-hand-side of `var` definitions and assignments are always atomic. The range of the transformation is denoted as $Source_{ANF}$. To illustrate ANF in action, consider the following expression.

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

The resulting block directly encodes the data-flow dependencies of the original program.

$$\{\ \mathtt{val\ u_1} = \mathtt{x} * \mathtt{x};\ \mathtt{val\ u_2} = \mathtt{y} * \mathtt{y};\ \mathtt{val\ u_3} = \mathtt{u_1} + \mathtt{u_2};\ \mathtt{z} = \mathtt{u_3};\ \mathtt{val\ u_4} = \mathtt{Math.sqrt(z)}\ ;\ \mathtt{u_4}\ \}$$

$$\text{ANF-ATOM}\ \frac{}{a \mapsto \{\ a\ \}} \qquad \frac{t \mapsto \{\ ss;\ a\ \}}{\{\ t\ \} \mapsto \{\ ss;\ a\ \}}\ \text{ANF-BLCK}$$

$$\text{ANF-ASCR}\ \frac{t \mapsto \{\ ss;\ a\ \}}{t:\ T \mapsto \{\ ss; \mathtt{val}\ x = a:\ T;\ x\ \}} \qquad \text{ANF-FUN}\ \frac{t \mapsto t'}{pdefs \Rightarrow t \mapsto pdefs \Rightarrow t'}$$

$$\text{ANF-IF}\ \frac{t_1 \mapsto \{\ ss;\ a\ \} \qquad t_2 \mapsto t_2' \qquad t_3 \mapsto t_3'}{\mathtt{if}\ (t_1)\ t_2\ \mathtt{else}\ t_3 \mapsto \{\ ss; \mathtt{val}\ x = \mathtt{if}\ (a)\ t_2'\ \mathtt{else}\ t_3';\ x\ \}}$$

$$\text{ANF-MOD}\ \frac{t \mapsto \{\ ss;\ a\ \}}{t.module \mapsto \{\ ss; \mathtt{val}\ x = a.module;\ x\ \}}$$

$$\text{ANF-CALL}\ \frac{t \mapsto \{\ ss;\ a\ \} \qquad \forall t_{jk}.\ t_{jk} \mapsto \{\ ss_{jk};\ a_{jk}\ \}}{t.m[T_i](t_{jk})_j \mapsto \{\ ss; ss_{jk}; \mathtt{val}\ x = a.m[T_i](a_{jk})_j;\ x\ \}}$$

$$\text{ANF-NEW}\ \frac{\forall t_{jk}.\ t_{jk} \mapsto \{\ ss_{jk};\ a_{jk}\ \}}{\mathtt{new}\ C[T_i](t_{jk})_j \mapsto \{\ ss_{jk}; \mathtt{val}\ x = \mathtt{new}\ C[T_i](a_{jk})_j;\ x\ \}}$$

$$\text{ANF-VAL}\ \frac{t_1 \mapsto \{\ ss_1;\ a_1\ \} \qquad \{\ ss;\ t_2\ \} \mapsto \{\ ss_2;\ a_2\ \}}{\{\ \mathtt{val}\ x = t_1; ss;\ t_2\ \} \mapsto \{\ ss_1; [x := a_1] ss_2;\ [x := a_1] a_2\ \}}$$

$$\text{ANF-VAR}\ \frac{t_1 \mapsto \{\ ss_1;\ a_1\ \} \qquad \{\ ss;\ t_2\ \} \mapsto \{\ ss_2;\ a_2\ \}}{\{\ \mathtt{var}\ x = t_1; ss;\ t_2\ \} \mapsto \{\ ss_1; \mathtt{var}\ x = a_1; ss_2;\ a_2\ \}}$$

$$\text{ANF-ASGN}\ \frac{t_1 \mapsto \{\ ss_1;\ a_1\ \} \qquad \{\ ss;\ t_2\ \} \mapsto \{\ ss_2;\ a_2\ \}}{\{\ x = t_1; ss;\ t_2\ \} \mapsto \{\ ss_1; x = a_1; ss_2;\ a_2\ \}}$$

$$\text{ANF-LOOP}\ \frac{loop \mapsto loop' \qquad \{\ ss;\ t\ \} \mapsto \{\ ss';\ a\ \}}{\{\ loop; ss;\ t\ \} \mapsto \{\ loop'; ss';\ a\ \}}$$

$$\text{ANF-WDO}\ \frac{t \mapsto t' \qquad block \mapsto block'}{\mathtt{while}\ (t)\ block \mapsto \mathtt{while}\ (t')\ block'}$$

$$\text{ANF-DOW}\ \frac{t \mapsto t' \qquad block \mapsto block'}{\mathtt{do}\ block\ \mathtt{while}\ (t) \mapsto \mathtt{do}\ block'\ \mathtt{while}\ (t')}$$

Figure 6.2: Inference rules for the ANF transformation.

$$\text{DSCF-REF1}\ \frac{x \notin \mathcal{V}}{\mathcal{V} \vdash x \mapsto x} \qquad \text{DSCF-REF2}\ \frac{\mathcal{V}x = a}{\mathcal{V} \vdash x \mapsto a} \qquad \text{DSCF-ASCR}\ \frac{\mathcal{V} \vdash a \mapsto a'}{\mathcal{V} \vdash a : T \mapsto a' : T} \qquad \text{DSCF-MOD}\ \frac{\mathcal{V} \vdash a \mapsto a'}{\mathcal{V} \vdash a.module \mapsto a'.module}$$

$$\text{DSCF-FUN}\ \frac{\mathcal{V} \vdash block \mapsto let}{\mathcal{V} \vdash pdefs \Rightarrow block \mapsto pdefs \Rightarrow let} \qquad \frac{\mathcal{V} \vdash a \mapsto a' \qquad \mathcal{V} \vdash \forall a_{jk}.\ a_{jk} \mapsto a'_{jk}}{\mathcal{V} \vdash a.m[T_i](a_{jk})_j \mapsto a'.m[T_i](a'_{jk})_j}\ \text{DSCF-CALL}$$

$$\text{DSCF-NEW}\ \frac{\mathcal{V} \vdash \forall a_{jk}.\ a_{jk} \mapsto a'_{jk}}{\mathcal{V} \vdash \texttt{new}\ C[T_i](a_{jk})_j \mapsto \texttt{new}\ C[T_i](a'_{jk})_j} \qquad \frac{}{\mathcal{V} \vdash \{\ kdefs;\ c\ \} \mapsto \{\ kdefs;\ c\ \}}\ \text{DSCF-LET}$$

$$\text{DSCF-VAL}\ \frac{\mathcal{V} \vdash b \mapsto b' \qquad \mathcal{V} \vdash \{\ ss;\ c\ \} \mapsto \{\ vdefs;\ kdefs;\ c'\ \}}{\mathcal{V} \vdash \{\ \texttt{val}\ x = b; ss;\ c\ \} \mapsto \{\ \texttt{val}\ x = b'; vdefs;\ kdefs;\ c'\ \}}$$

$$\text{DSCF-VAR}\ \frac{\mathcal{V} \vdash a \mapsto a' \qquad \mathcal{V}, x \leftarrow a' \vdash \{\ ss;\ c\ \} \mapsto let}{\mathcal{V} \vdash \{\ \texttt{var}\ x = a; ss;\ c\ \} \mapsto let} \qquad \frac{}{\mathcal{V} \vdash lit \mapsto lit}\ \text{DSCF-LIT}$$

$$\text{DSCF-ASGN}\ \frac{\mathcal{V} \vdash a \mapsto a' \qquad \mathcal{V}, x \leftarrow a' \vdash \{\ ss;\ c\ \} \mapsto let}{\mathcal{V} \vdash \{\ x = a; ss;\ c\ \} \mapsto let} \qquad \frac{}{\mathcal{V} \vdash this \mapsto this}\ \text{DSCF-THIS}$$

$$\text{DSCF-IF1}\ \frac{\begin{array}{c} x_i \in (\mathcal{A}[\![\{\ ss_1;\ a_1\ \}]\!] \cup \mathcal{A}[\![\{\ ss_2;\ a_2\ \}]\!]) \cap \mathcal{R}[\![\{\ ss_3;\ c_3\ \}]\!] \\ \mathcal{V}x_i = a'_i \qquad x \in \mathcal{R}[\![\{\ ss_3;\ c_3\ \}]\!] \qquad \mathcal{V}, x \leftarrow p, x_i \leftarrow p_i \vdash \{\ ss_3;\ c_3\ \} \mapsto let_3 \\ \mathcal{V} \vdash \{\ ss_1;\ k_3(a_1, x_i)\ \} \mapsto let_1 \qquad \mathcal{V} \vdash \{\ ss_2;\ k_3(a_2, x_i)\ \} \mapsto let_2 \end{array}}{\begin{array}{l} \mathcal{V} \vdash \{\ \texttt{val}\ x = \texttt{if}\ (a)\ \{\ ss_1;\ a_1\ \}\ \texttt{else}\ \{\ ss_2;\ a_2\ \}; ss_3;\ c_3\ \} \\ \quad \mapsto \{\ \texttt{def}\ k_1() = let_1; \texttt{def}\ k_2() = let_2; \texttt{def}\ k_3(p, p_i) = let_3;\ \texttt{if}\ (a)\ k_1()\ \texttt{else}\ k_2()\ \} \end{array}}$$

$$\text{DSCF-IF2}\ \frac{\begin{array}{c} x_i \in (\mathcal{A}[\![\{\ ss_1;\ a_1\ \}]\!] \cup \mathcal{A}[\![\{\ ss_2;\ a_2\ \}]\!]) \cap \mathcal{R}[\![\{\ ss_3;\ c_3\ \}]\!] \\ \mathcal{V}x_i = a'_i \qquad x \notin \mathcal{R}[\![\{\ ss_3;\ c_3\ \}]\!] \qquad \mathcal{V}, x_i \leftarrow p_i \vdash \{\ ss_3;\ c_3\ \} \mapsto let_3 \\ \mathcal{V} \vdash \{\ ss_1;\ k_3(x_i)\ \} \mapsto let_1 \qquad \mathcal{V} \vdash \{\ ss_2;\ k_3(x_i)\ \} \mapsto let_2 \end{array}}{\begin{array}{l} \mathcal{V} \vdash \{\ \texttt{val}\ x = \texttt{if}\ (a)\ \{\ ss_1;\ a_1\ \}\ \texttt{else}\ \{\ ss_2;\ a_2\ \}; ss_3;\ c_3\ \} \\ \quad \mapsto \{\ \texttt{def}\ k_1() = let_1; \texttt{def}\ k_2() = let_2; \texttt{def}\ k_3(p_i) = let_3;\ \texttt{if}\ (a)\ k_1()\ \texttt{else}\ k_2()\ \} \end{array}}$$

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

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

Figure 6.3: Inference rules for the DSCF transformation.

The subsequent translation from $Source_{ANF}$ to $Core_{ANF}$ is handled by the DSCF transformation (Figure 6.3). For *Source* terms t without `var` definitions, assignments, and control-flow, DSCF will simply map the *stats* blocks in ANF$[\![t]\!]$ to $Core_{ANF}$ *let* blocks. To eliminate variables, the transformation relies on an environment $\mathcal{V}$ that keeps track of the most recent atomic term a associated with each variable x. The environment is updated in rules DSCF-VAR and DSCF-ASGN and accessed in rule DSCF-REF2. Loop constructs and conditional terms are translated by rules DSCF-IF1, DSCF-IF2, DSCF-WDO, and DSCF-DOW. 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. Variables x_i assigned in the matched control-flow structure are converted to parameters p_i in the corresponding continuation definitions. Rules DSCF-IF1 and DSCF-IF2 handle a conditional term of the form

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

and diverge depending on whether $x \in \mathcal{R}[\![\{\ ss_3;\ c_3\ \}]\!]$ or not. If x is referenced in the suffix, the signature and the calls of the corresponding continuation k_3 have to be adapted accordingly.

The DSCF rewrite also ensures certain properties of the resulting trees. First, the dominator tree of the control-flow graph is encoded by the parent-child relationship of the nested continuation function definitions. Second, lexical scope is preserved – continuation functions do not have parameters that always bind to the same argument. As these properties are commonly assumed by compiler optimizations, this alleviates the need of a dedicated lambda dropping rewrite [DS00]. Third, excluding terms in nested lambda bodies, the resulting term has exactly one *let* block of the form { vals ; a } which we denote as SUFFIX$[\![t]\!]$.

6.2 First-Class Monad Comprehensions

An IR for *Emma* should accommodate common optimizations from both the language and the query compilation domains. *Emma* $Core_{ANF}$ is good fit for the first, but too

$b :=$	...	binding term	$stat :=$	...	statement
	`for` { *qs* } `yield` *let*	comprehension		q	qualifier
			$q :=$		qualifier
				$x \leftarrow$ *let*	generator
				`if` *let*	guard

Figure 6.4: Extending the abstract syntax of *Emma* $Core_{ANF}$ to *Emma Core*.

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

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

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

Figure 6.5: Inference rules for the RESUGAR$_{\mathsf{M}}$ transformation. The type former M should be a monad, i.e., it should implement `map`, `flatMap`, and `withFilter` obeying the "monad with zero" laws.

low level for the second. Query compilation usually starts with queries expressed in a `Select-From-Where`-like concrete syntax and translates to a relational algebra expression. To that end, most commonly one uses a join graph extracted from the `Select-From-Where` expression as a basis for join-order enumeration based on dynamic programming [SAC+79, GLSW93, MN06]. In line of the similarities between `Select-From-Where` expressions and `for`-comprehensions outlined in Section 5.3.2, our goal is to enable similar techniques on *Emma* `Bag` comprehensions. Unfortunately, traditional ANF forms such as $\mathit{Core}_{\mathit{ANF}}$ encode `for`-comprehensions in their desugared form, i.e., as chains of nested `flatMap`, `withFilter`, and `map` operators. To overcome this limitation, we extend $\mathit{Core}_{\mathit{ANF}}$ with support for first-class monad comprehension syntax.

The resulting extended language, called *Emma Core*, is depicted on Figure 6.4. Observe that, similar to lambdas, sub-terms in the new syntactic forms – comprehension head, generator right-hand-side, and guard expression – are restricted to be *let* blocks. This constraint simplifies definitions on top of *Emma* without loss of expressive power – a terms expand to $\{\ \ a\ \}$ and b terms to $\{\ \ \texttt{val}\ x = b\ ;\ \ x\ \}$.

The translation from *Emma* $\mathit{Core}_{\mathit{ANF}}$ to *Emma Core* proceeds in two steps. First, we apply the RESUGAR$_{\texttt{Bag}}$ transformation, which converts `flatMap`, `withFilter`, and `map` calls on `Bag` targets to simple monad comprehensions. Figure 6.5 lists the main inference rules of RESUGAR. Application of these rules depends on a context $\mathcal{X}$ of available lambda definitions. Due to space considerations, the rules that accumulate $\mathcal{X}$ and eliminate the monad operator applications if `f` is not in $\mathcal{X}$ are omitted from the figure. Informally, the transformation operates as if $\mathcal{X}$ is defined as follows.

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

In other words, if f is not available in the current scope, $\mathcal{X}$ associates f with an eta-expansion of itself – that is, with a function that just applies f. This allows to resugar not only terms representing desugared `for`-comprehensions, but also *Emma Source* terms defined directly in desugared form, as for example `xs.withFilter(isPrime)` where `isPrime` is not defined in the quoted code fragment.

6.3 Comprehension Normalization

Upon applying the RESUGAR$_{\texttt{Bag}}$ transformation, we proceed with a normalization step that repeatedly constructs bigger comprehensions by merging def-use chains of smaller ones. The benefits of this process are motivated by the targeted query compilation techniques – optimizing bigger comprehensions improves the chances of producing better execution plans.

The normalizing transformation NORMALIZE$_{\mathsf{M}}$ consists of a single rule – UNNEST-HEAD, which is applied repeatedly until convergence. The rule is depicted on Figure 6.6. The consequent matches an enclosing *let* block which 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 identifies a comprehension which is defined in $vdefs_1$ or $vdefs_2$ and is referenced only once (in the x_1 definition). The rewrite depends on the auxiliary functions *split*, *fix* and *remove* which operate as follows. First,

$$remove(x, vdefs)$$

removes a value definition $\texttt{val}\ x = b$ from *vdefs*. Second,

$$split(vdefs, qs)$$

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,

$$fix(e)$$

where $e = x \leftarrow let \mid \texttt{if}\ let \mid let$ adapts $let = \{\ vals;\ defs;\ c\ \}$ in two steps. First, it obtains let' by inlining let_2 which defines x_3 in *let*. If $x_3 \notin \mathcal{R}[\![let]\!]$, we have $let' = let$, otherwise let' is derived from let_2 by extending the suffix SUFFIX$[\![let_2]\!] = \{\ vals_S;\ a_S\ \}$ as $[x_3 := a_S]\{\ vals_S; vals;\ defs;\ c\ \}$. Second, copies of the dependent values $vdefs_2^D$ that

UnnestHead

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

Figure 6.6: The UNNEST-HEAD rule used in the NORMALIZE$_{\mathsf{M}}$ transformation. As in Figure 6.5, M can be any type former which is also a monad.

are referenced in *let* are prepended to *let′*.

6.4 Binding Context

The `Bag` abstraction presented in Section 5.3 allows for nesting. A nested `Bag` can be constructed either as a result of a `groupBy` application or directly, e.g. by the following expression.

```
val xs : Bag[Bag[String]] = for { d ← docs } yield tokenize(d)
```

While this leads to a more unified programming model, it poses some challenges at compile-time. The goal of the *Emma* compiler is to execute `Bag` expressions on a parallel dataflow engine by implementing them in an engine-specific API such as Spark's `RDD` or Flink's `DataSet`. Due to the limitations identified in Chapter 3, however, the target APIs lack the nesting support of our `Bag` abstraction. To write the above expression in Spark's `RDD` API, for example, one has to change the return type of `tokenize` to `Seq[String]` instead of `RDD[String]`.

```
val xs : Bag[Seq[String]] = for { d ← docs } yield tokenize(d)
```

A naïve type-directed translation scheme which implements *all* `Bag` terms in the target API therefore is not a feasible compilation strategy as it might lead to runtime errors. Instead, we want to translate only those `Bag` expressions that occur at the top level – that is, those that are not nested within other `Bag` expressions. To achieve that, we have to estimate the *binding context* of all symbols.

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

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

(b) computed binding context values

Figure 6.7: Binding context example.

To determine $\mathcal{C}(x)$ for all binding symbols x defined in an *Emma Core* term t we use a procedure called CONTEXT$[\![t]\!]$. To illustrate how CONTEXT works, consider the example from Figure 6.7a. We first define a function `f` which extracts `brands` mentioned in a document `doc`. Upon that, we use `f` to compute the `Bag` of brands `bs` mentioned in a seed document `d0`. Finally, from the `Bag` of documents `docs` we select only those documents `d` which mention a brand `b` also contained in `bs`.

Figure 6.7b depicts the result of CONTEXT procedure for this example snippet. The binding context of symbols defined in the outer-most scope (such as `f`, `bs`, and `rs`) is always *Driver*. The binding context of symbols representing comprehension generators (such as `d` and `b`) is always *Engine*. The binding context of symbols nested in lambdas, however, depends on the lambda uses. In our running example, `f` is used both in a *Driver* context (in the `bs` definition), as well as in an *Engine* context (in the `rs` definition). Consequently, the binding context of all symbols defined in the `f` lambda (such as `doc` and `brands`) is *Ambiguous*. The context of nested lambdas can be computed recursively.

We want to specialize all definitions of terms which denote a `Bag` constructor application and are evaluated in the driver. As a conservative approach, we therefore prohibit programs where such terms have *Ambiguous* binding context. In our running example, compilation will fail because $\mathcal{C}(\texttt{brands}) = \textit{Ambiguous}$. To alleviate this restriction, one can employ a more elaborate strategy that duplicates lambdas with ambiguous use (such as `f`) and disambiguates their use sites. In practice, however, the data analysis programs we analyzed and implemented so far did not suffer from this issue, so we opted for the more restrictive, but simpler approach.

6.5 Compiler Pipelines

Putting the pieces together, we can now sketch the high-level layout of all compilation pipelines realized on top of the *Emma* compiler infrastructure. The transformations

presented so far form the basis of a generic compiler frontend which is defined as follows.

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

Quoted *Emma Source* terms are first lifted from *Emma Source* to *Emma Core* by the LIFT pipeline. The resulting term is then iteratively transformed by a chain of *Core* ⇒ *Core* optimizing transformations such as the ones discussed in Section 7.1 through Section 7.4.

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

The specific optimizing transformations might be defined either in a backend-agnostic or a backend-specific 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). Nevertheless, the OPTIMIZE chain always ends with a transformation that specializes the backend. This transformation identifies *vdef* terms of the form `val` x = `Bag`.$m[\ldots](\ldots)$, where $\mathcal{C}(x) = \textit{Driver}$ and m matches one of the source methods listed in Figure 5.2. The `Bag` companion object is then substituted either with `SparkBag` or `FlinkBag`, depending on the desired backend.

Finally, we apply an inverse DSCF transformation that "lowers" the resulting code to an executable form. Continuation definitions are thereby converted back to control-flow statements (such as `while` loops), and continuation parameters are converted back to variable definitions and variable assignments. This is necessary in order to avoid stack overflow errors at runtime, as in contrast to other functional programming languages, Scala eliminates tail calls only in self-recursive methods.

We end up with two different basic pipelines defined as Scala macros – one for Spark and one for Flink. Scala code that is "quoted" (that is, enclosed) in one of these macros is transformed by the corresponding pipeline. Optimizing transformations in the pipeline can be switched off and on with an optional configuration. The following snippet illustrates the use of *Emma* macros.

```
onSpark("noCache.conf") {
  val ds = Bag.readCSV[Doc](...)
  val ps = tfidf(d)
  ...
}
```

```
onFlink {
  val ds = Bag.readCSV[Doc](...)
  val ps = tfidf(d)
  ...
}
```

The snippet on the left compiles the enclosed code for Spark with a customized OPTIMIZE pipeline that excludes automatic `cache` insertion. The snippet on the right compiles the same code for Flink using the default Flink pipeline. The `ds` *vdef* is specialized accordingly in each case.

In order to facilitate modularity, we also add a `lib` macro-annotation which can be

used to annotate objects containing library functions. Quoted calls to such functions are (recursively) inlined before applying the LIFT frontend, and lambdas used only once in a direct application are β-reduced before the OPTIMIZE step. In the above example, the `tfidf` is a library method that calls another library method called `tokenize`, so both methods are first inlined in the enclosing code snippet. This mechanism enables authors to write modular and composable libraries based on *Emma* without impeding the optimization potential of quoted code fragments in which these libraries are used.

7 Optimizations

Having established *Emma Core* as an IR for our embedded DSL, we can now demonstrate its utility for on a variety of enabled optimizations. Section 7.1 discusses a compilation scheme that translates comprehension syntax terms into parallel dataflows comprised of binary combinators such as `equiJoin` and `cross`. Section 7.2 discusses an optimization which reduces the data-shuffle footprint of an application through automatic insertion of primitives for partial aggregates. Section 7.3 outlines a strategy for automatic insertion of `cache` calls. Finally, Section 7.4 presents a Flink-specific optimization which introduces specialized `iterate` calls for certain types of `while` loops.

7.1 Comprehension Compilation

The *Emma Core* language presented in Chapter 6 resugars applications of `Bag` monad operators as `Bag` comprehensions, normalizes those, and integrates the result as first-class syntax in the *Emma Core* IR. The *Emma* compiler then has to rewrite the normalized `Bag` comprehensions as dataflow expressions based on the operators supported by the targeted parallel dataflow engines.

7.1.1 Naïve Approach

A naïve approach is to adopt Scala's $\text{DESUGAR}_{\texttt{Bag}}$ scheme (see **F6** from Section 5.1). Unfortunately, this strategy can easily produce suboptimal dataflows. To illustrate why, let e denote a comprehension defining an equi-join between two `Bag` terms xs and ys.

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

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

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

A subsequent specialization of *xs* to a `FlinkBag` or `SparkBag` will parallelize the application of the `flatMap` operator. However, the `withFilter` and `map` calls on *ys* are nested inside the already parallelized `flatMap` lambda. The resulting dataflow therefore acts like a broadcast nested-loop join where *ys* corresponds to the inner (broadcast) and *xs* to the outer (partitioned) relation.

7.1.2 Qualifier Combination

As we saw in the Flink and Spark examples listed in Section 2.3, the parallel dataflow engines we target support efficient distributed equi-joins via dedicated operators. To utilize those, we adopt the approach of Grust [GS99, Gru99] 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 bundled in a `ComprehensionCombinators` trait and implemented three times. The `LocalOps` implementation uses the above naïve definitions, whereas `FlinkOps` and `SparkOps` directly apply the corresponding native operator on the backing distributed collection. For example, assuming that the backing Flink `DataSet` of a `FlinkBag` `xs` can be extracted with an `asDataSet(xs)` call, `equiJoin` can be defined in `FlinkOps` 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).rep join asDataSet(ys).rep) where kx equalTo ky)
)
```

Based on these combinators, we design a rule-based comprehension compilation strategy called COMBINE (the rules are depicted on Figure 7.1). In addition to rules COM-JOIN and COM-CROSS, we add 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 `withFilter` rule comes in two flavors – COM-FMAP2 is applied if the eliminated generator variable x is referenced in subsequent terms, while COM-FMAP1 is applied otherwise.

The rules rely on the following auxiliary functions. $\mathcal{R}[\![t]\!]$ denotes the set of symbols referenced by t (as in Figure 6.3), and $\mathcal{R}^*[\![t]\!]$ the ones upon which t transitively depends.

COM-FILTER

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

COM-FMAP1

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

COM-FMAP2

$$\frac{x \in \mathcal{R}[\![ys]\!] \qquad \mathcal{R}[\![ys]\!] \cap \mathcal{G}[\![qs_2]\!] = \varnothing \qquad t' = [x := z.\_1][y := z.\_2]t}{\texttt{for}\ \{\ qs_1;\ x \leftarrow xs;\ qs_2;\ y \leftarrow ys;\ qs_3\ \}\ \texttt{yield}\ let \mapsto \texttt{for}\ \{\ qs_1;\ z \leftarrow xs.\texttt{flatMap}(x \Rightarrow ys.\texttt{map}(y \Rightarrow (x, y)));\ qs'_2;\ qs'_3\ \}\ \texttt{yield}\ let'}$$

COM-JOIN

$$\frac{x \notin \mathcal{R}^*[\![ys]\!] \qquad x \in \mathcal{R}[\![k_x]\!] \qquad x \in \mathcal{R}[\![k_u]\!] \qquad t' = [x := z.\_1][y := z.\_2]t \qquad \mathcal{R}[\![k_y]\!] \cap \mathcal{G}[\![qs_1 +\!\!+ qs_2]\!] = \varnothing \qquad \mathcal{R}[\![k_x]\!] \cap \mathcal{G}[\![qs_1 +\!\!+ qs_2]\!] = \varnothing}{\texttt{for}\ \{\ qs_1;\ x \leftarrow xs;\ qs_2;\ y \leftarrow ys;\ qs_3;\ \texttt{if}\ k_x = k_y;\ qs_4\ \}\ \texttt{yield}\ let \mapsto \texttt{for}\ \{\ qs_1;\ z \leftarrow \texttt{equiJoin}(x \Rightarrow k_x, y \Rightarrow k_y)(xs, ys);\ qs'_2;\ qs'_3\ \}\ \texttt{yield}\ let'}$$

COM-CROSS

$$\frac{x \notin \mathcal{R}^*[\![ys]\!] \qquad x \in \mathcal{R}[\![k_x]\!] \qquad x \in \mathcal{R}[\![k_u]\!] \qquad t' = [x := z.\_1][y := z.\_2]t \qquad \mathcal{R}[\![k_y]\!] \cap \mathcal{G}[\![qs_1 +\!\!+ qs_2]\!] = \varnothing \qquad \mathcal{R}[\![k_x]\!] \cap \mathcal{G}[\![qs_1 +\!\!+ qs_2]\!] = \varnothing}{\texttt{for}\ \{\ qs_1;\ x \leftarrow xs;\ qs_2;\ y \leftarrow ys;\ qs_3\ \}\ \texttt{yield}\ let \mapsto \texttt{for}\ \{\ qs_1;\ z \leftarrow \texttt{cross}(xs, ys);\ qs'_2;\ qs'_3\ \}\ \texttt{yield}\ let'}$$

COM-MAP

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

Figure 7.1: Rules introducing comprehension combinators as part of the COMBINE transformation.

$\mathcal{G}[\![qs]\!]$ denotes the set of generator symbols bound by the qualifier sequence qs. For example, the premises of COM-FILTER state that p should reference x, but should not reference any symbols bound by generators in qs_1 or qs_2.

Note that the presentation in Figure 7.1 is simplified, as the actual implementations maintain *Emma Core* form. For example, instead of xs, COM-FILTER actually matches a let_{xs} term with

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

and rewrites $\text{SUFFIX}[\![let_{xs}]\!]$ using fresh symbols `f` and `y` as follows.

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

The COMBINE scheme iteratively applies the first matching rule. The specific rule order indicated on Figure 7.1 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 aligns with common heuristics exploited by rule-based query optimizers [Fre87]. To illustrate the 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)
```

Normalization proceeds in three steps. In the first two, the COMB-JOIN rule combines `m` and `c` (introducing `u`), and then `u` and `p` (introducing `v`). The intermediate results look as follows.

```
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, the COMB-MAP rule 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 scheme is complemented by an extension of the `Bag` specialization procedure outlined in Section 6.5. In addition to `Bag` companion constructors, we also specialize combinator applications, replacing `LocalOps` with either `FlinkOps` or `SparkOps` depending on the selected backend.

7.1.3 Structured API Specialization in Spark

The COMBINE transformation uses established query processing heuristics in order to translate `Bag` comprehensions as parallel dataflows targeting Flink or Spark. The resulting dataflow graphs are then further optimized by the target engine. Both engines automatically fuse operators that can be executed in a single pass (e.g., a chain of `map` and `withFilter` applications). In addition, Flink's built-in optimizer automatically selects optimal data distribution and local execution strategies for operators such as `cross` and `equiJoin`. To enable similar functionality in Spark, however, one has to express the target dataflows in Spark's structured `Dataset` API. To achieve this automatically, we extend Spark's OPTIMIZE pipeline with a corresponding specializing transformation.

The transformation proceeds in two steps. First, we identify lambdas used in dataflow operators backed by Spark, and for each lambda, we check whether its definition can be specialized as a corresponding Spark `Column` expression. In the second step, we specialize dataflow operators whose lambdas can be specialized. Below, we sketch these steps based on our running example.

We model the set of supported Spark `Column` expressions as an `Expr` ADT equipped with an evaluator function `eval` : `Expr` ⇒ `Column`. Lambda specialization is restricted to lambdas without control-flow and preserves the ANF structure of the lambda body. More specifically, for each *vdef* in the *let* block constituting the lambda body, we check whether its right-hand-side can be mapped to a corresponding `Expr`. If this is true for all *vdefs*, we can specialize the lambda, changing its type from `A` ⇒ `B` to `Expr` ⇒ `Expr`. To illustrate this process, consider the top-level join of the dataflow depicted at the end of Section 7.1.2. The `u` ⇒ `u._2.personID` lambda is specialized as follows (showing the *Emma Core* version 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
}
```

All other lambdas in the example dataflow can be specialized in a similar way. Consequently, the `equiJoin` and `map` applications using these lambdas can be specialized as well. To that end we define an object `SparkNtv` with specialized dataflow operators `equiJoin`, `select`, and `project` corresponding to `equiJoin`, `map`, and `withFilter`. For

example, `equiJoin` is defined as follows.

```
def equiJoin[A, B, K](
  kx: Expr => Expr, ky: Expr => Expr)(xs: Bag[A], ys: Bag[B]
): Bag[(A, B)] = {
  val (us, vs) = (asDataset(xs), asDataset(ys))
  val cx = eval(kx(Root(us)))
  val cy = eval(ky(Root(vs)))
  SparkBag(us.joinWith(vs, cx === cy))
}
```

The implementation accepts the original bags `xs` and `ys` next to the specialized lambdas `kx` and `ky`. We first extract the `Dataset` representations of the two input bags. We then use those to evaluate the specialized lambdas and obtain `Column` expressions for the corresponding join keys. Finally, we construct a Spark `Dataset` equi-join and wrap the result in a new `SparkBag`.

The presented approach ensures that we implement *Emma* dataflows on Spark in terms of the more efficient, optimizable `Dataset` API whenever possible, and in terms of the more general `RDD` API otherwise. The strategy is also more future-proof than writing hard-coded Spark dataflows. When a new Spark version rolls out, we only need to add support for the new `Column` expressions to the lambda specialization logic. Clients can then re-compile their *Emma Source* code without client-side code modifications, and benefit from the larger dataflow fragments compiled to the Spark `Dataset` API.

7.2 Fold Fusion

The FOLD-FUSION optimization presented in this section resolves the issues outlined in Example 2.3 and is facilitated by the following *Emma* design aspects. First, the `Bag` API is derived from a solid algebraic foundation, using UNION-representation as a model for distributed data and its associated structural recursion operator (`fold`) as a model for parallel collection processing (see Section 4.1). Second, the API allows for nested computations – the `groupBy` method transforms a `Bag`A into a `Bag` of `Group` instances where each group contains a `values` member of type `Bag`A (see Section 5.3.3). Third, the quotation-based compilation approach allows for representing such nested computations in *Emma Core* and designing algebraic rewrites based on this holistic IR.

Internally, the FOLD-FUSION optimization is defined as the composition of two rewrites

$$\text{FOLD-GROUP-FUSION} \circ \text{FOLD-FOREST-FUSION} \quad .$$

We discuss each rewrite in detail. As a running example, consider a code snippet which computes `min` and `avg` values per group from a `Bag` of data 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)
}
```

7.2.1 Fold-Forest Fusion

The goal of FOLD-FOREST-FUSION is to rewrite a tree of folds over different UNION-algebras as a single fold over a corresponding tree of UNION-algebras. The rewrite proceeds in three steps.

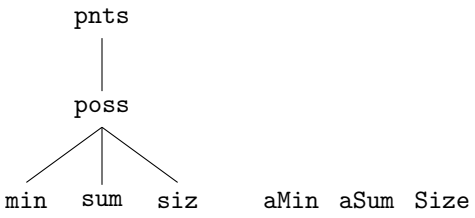
Fold Inlining and Fold-Forest Construction

As a first step, we inline all aliased folds and extract a forest of of `fold` applications. Each tree in the forest is rooted in a different `Bag` instance. Leaf nodes in the tree represent `fold` applications. Inner nodes represent linear `Bag` comprehensions, i.e. comprehensions of the general form (omitting possibly occurring guards)

$$\texttt{for}\ \{\ x_1 \leftarrow let_1\ ;\ldots;\ x_n \leftarrow let_n\ \}\ \texttt{yield}\ let_h$$

where each generator references the symbol bound from the previous one, i.e. $\forall 1 \leqslant i < n:\ x_i \in \mathcal{R}[\![let_{i+1}]\!]$. In our running example, the definitions of the `stat.min` and `stat.avg` folds are expanded (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)
}
```



Trees are then collapsed in a bottom-up way by a FOLD-FOREST-FUSION rewrite, realized as an interleaved application of two rewrite rules. The BANANA-FUSION rewrite merges leaf siblings into a single leaf, whereas CATA-FUSION merges an inner node which has a single leaf as a child.

Banana-Fusion

The rewrite is enabled by the Banana-Split law from Section 4.1.7, which states that any pair of folds can be fused into a single fold over a pair of algebras, i.e.

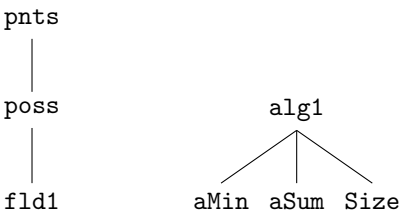
$$(xs.\texttt{fold}(alg_1), xs.\texttt{fold}(alg_2)) = xs.\texttt{fold}(\texttt{Alg2}(alg_1, alg_2))$$

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))
}
```

The law generalizes to n-ary tuples, which means that with a single application from left to right of the above equation we can "fuse" leafs sharing a common parent. In our running example, we first fuse the `aMin`, `aSum`, and `Size` algebras as `alg1`, and then fuse the corresponding `min`, `sum` and `siz` folds as `fld1`. The three leafs of the original fold tree thereby collapse into a single leaf (on the left). The original structure is now mirrored in the tree of Union-style algebras (on the right).

```
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
...
```



Cata-Fusion

The rewrite is inspired by the Cata-Map-Fusion law from Section 4.1.7, which states that a fold over a recursive datatype can be fused with a preceding `map`f application.

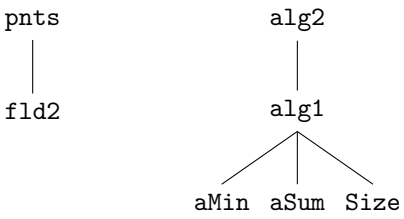
$$xs.\texttt{map}(f).\texttt{fold}(a) = xs.\texttt{fold}(\texttt{AlgMap}(f, a))$$

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 = f andThen a.init
  val plus = a.plus
}
```

In our running example, `poss` is defined as 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 can therefore apply the *cata-fusion* law directly in order to fuse `poss` with `fld1`. The final result looks as follows.

```
...
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
...
```



Observe the symmetry between the original tree of folds the the resulting three of algebras.

Based on the insight that all `Bag` comprehensions admit a catamorphic interpretation [Gru99], we extend the CATA-FUSION rewrite with two more algebras which allow for fusing arbitrary linear comprehensions. Folds $ys.\texttt{fold}(a)$ 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 thereby fused as $xs.\texttt{fold}(\texttt{AlgFilter}(p, a))$, 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 follows.

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

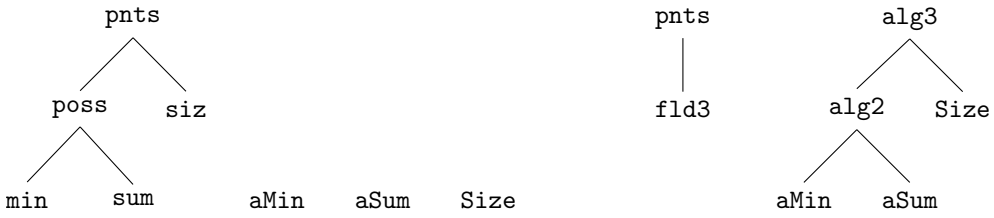
Similarly, folds $ys.\texttt{fold}(a)$ where ys is a linear comprehension of the general form defined in Section 7.2.1 are fused as $xs.\texttt{fold}(\texttt{AlgFlatMap}(f, a))$, where `AlgFlatMap` is defined as

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

and the argument f is constructed as follows.

$$\texttt{val}\, f = x_1 \Rightarrow \texttt{for}\ \{\ x_2 \leftarrow let_2\ ; \ldots ;\ x_n \leftarrow let_n\ \}\ \texttt{yield}\ let_h$$

The outlined fusion approach therefore works on fold trees with arbitrary shape. For example, consider a variation of our running example where the `siz` aggregate is defined not as `poss.fold(Size)` but as `pnts.fold(Size)`. The original fold and algebra trees (on the left) and the resulting tree (on the right) change their shape in the following way.



Compared to the original example, we fuse only two leafs (`aMin` and `aSum`) in the first step, and apply an additional BANANA-FUSION between `alg2` and `Siz` in order to construct the root of the resulting tree of algebras `alg3`.

7.2.2 Fold-Group Fusion

While the fold-forest fusion optimization ensures that multiple aggregates derived from the same `Bag` instance can be computed in a single pass, the fold-group fusion optimization discussed in this section fuses group `values` consumed by a single `fold` with a preceding `groupBy` operation which constructs the groups. Note that fold-forest fusion therefore enables a subsequent fold-group fusion in situations where the group `values` is consumed by multiple folds. In our running example, in Section 7.2.1 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)
}
```

The fold-group fusion rewrite matches `groupBy` applications (such as `ptgrs`) that are used only once and the use occurs in the right-hand-side of a `Bag` comprehension generator (as in `stats`). The rewrite is subject to two conditions. First, the `values` field bound from each group (`pnts`) must be used only once as a target of a `fold` application. Second, the algebra passed to the `fold` application (`alg2`) should not depend on any other values

bound by the enclosing comprehension (such as `label`). If these conditions are met, we can pull the *vdefs* which construct the algebra out of the enclosing comprehension and replace the `groupBy` with a `foldGroup` call. Our running example is rewritten as follows.

```
... // 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 comprehension combinators introduced in Section 7.1, the `foldGroup` operator is defined in a `RuntimeOps` trait and mixed into `LocalOps`, `SparkOps`, and `FlinkOps`. The subsequent specializing transformation replacing `LocalOps` with one of the other two implementations (as described in Section 7.1.2) enables targeting the right parallel dataflow primitives. For example, `SparkOps` can define `foldGroup` in terms of the `RDD` API `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
  }
```

7.3 Caching

The next optimization we consider is automatic `cache` call insertion. Recall that due to the type-based deep embedding strategy, the distributed collection types exposed by Spark and Flink are *lazy*. Consequently, the same applies for *Emma*-based `FlinkBag` and `SparkBag` terms which are backed by Flink and Spark distributed collections. To illustrate the issues arising from this observation, consider a more specific, *Emma*-based variation of the second code snippet 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))
}
...
```

The code reads a text corpus and converts it into a `Bag` of labeled `points`. Feature extraction is done by tokenizing the document contents into a "bag of words" and feature hashing the resulting representation with the help of the `encode.freq` *Emma* library function. The constructed `points` are then randomly assigned to one of `K` folds and used for `k`-fold cross-validation with a logistic regression model. The cross-validation `for`-loop thereby accesses the `kfolds Bag` in each iteration. If the code is enclosed in an `onSpark` or `onFlink` quote, `kfolds` will be respectively specialized either 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 snippets below provide two more examples where `Bag` instances have to be cached.

```
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 one of three part-of-speech taggers in order to compute `tags`. Since `docs` is referenced more than once, it makes sense to cache it. Note that `cache` call insertion should not be too aggressive. For example, even if we exclude `size` from the snippet, `docs` is still referenced more than once. However, in this case it is actually beneficial to avoid caching in order to pipeline the execution of `docs`, `tags` and `rslts` in a single operator chain. To capture these situations, we have to ensure that `docs` references in mutually exclusive control-flow blocks are counted only once.

The right example computes all `paths` of length `N` from a `Bag` of `edges`. In this scenario, caching the loop-invariant `edges Bag` is not too beneficial, as it will only amortize the cost of a single `readCSV` execution per loop. On the other side, the loop-dependent `Bag` `paths` represents 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. In order to ensure that the size of iteratively constructed dataflows does not depend on the loop variable, we should conservatively cache loop-dependent `Bag` instances such as `paths`.

To cover the three cases outlined above, we define an optimization ADD-CACHE-CALLS based on the *Emma Core* IR. The rewrite caches `Bag` instances x if one of the following three 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.

All three cases can be identified based on analysis of the control- and data-flow graphs and the dominance tree derived from the *Emma Core* representation. C1 corresponds to situations where

- x is a value referenced in a continuation k;
- k is part of a cycle $k_1, \ldots, k_n$ embedded in the derived control-flow graph;
- x is not defined in any of the k_i contained in the cycle.

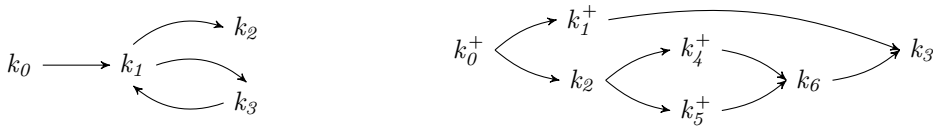
Further, let $uses_k(x)$ denote the number of uses for a symbol x in the continuation k (excluding uses in continuation calls), and let $dom(k)$ denote the set of continuations dominated by k (i.e., all continuation definitions nested in k). Then, C2 corresponds to situations where

- x is a value defined in a continuation k;
- the control-flow graph restricted to $dom(k)$ contains at least two strongly connected components S such that $\Sigma_{k_i \in S} uses_{k_i}(x) > 0$;
- at least two of the above components are also weakly connected between each other.

Finally, C3 corresponds to situations where

- x is a parameter of a continuation definition k,
- the 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.



The left graph corresponds to the snippets associated with C1[1] and C3. In both cases, the `for` loop is desugared as a `while` loop. The loop is represented by k_1 and its body by k_3. C1 applies to the first snippet, as

- `kfolds` is referenced in the k_3 continuation;

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

- k_3 is part of the k_1, k_3 cycle;
- `kfolds` is defined in $k_0 \notin \{k_1, k_3\}$.

In the third snippet, **C3** applies as

- the DSCF-WDO rule converts the variable `paths` as parameter of the k_1 continuation;
- the closure of the graph restricted to $dom(k_1) = \{k_1, k_3\}$ contains (k_1, k_1) as an edge.

The right graph corresponds to the code snippet illustrating **C2**. The superscript notation k^+ indicates that the value `docs` is referenced in the continuation k. **C2** applies to the third snippet, as

- `docs` is defined in k_0;
- the graph (without restrictions, because $dom(k_0) = \{k_0, \ldots, k_6\}$) 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`;
- from these candidate components, S_0 is connected with S_1, S_4 and S_5.

If we omit the `size` from k_0, the last condition is not met, as no pair from $\{S_1, S_4, S_5\}$ is connected.

The ADD-CACHE-CALLS optimization is backend-agnostic. The `cache` method is defined in the `RuntimeOps` trait and inserted as `LocalOps.cache` calls. These are then specialized in the same way as comprehension combinator and FOLD-FUSION calls (see Section 7.1.2 and Section 7.2): depending on the enclosing quote, `LocalOps` is replaced either by `FlinkOps` or `SparkOps`.

7.4 Native Iterations

As final optimization, we discuss a specializing transformation called SPECIALIZE-LOOPS, which maps *Emma Core* loops to Flink `iterate` operator calls. As discussed in Example 2.4, in contrast to Spark, Flink lacks full-fledged support for multi-dataflow applications. If the driver application has control flow and wants to execute multiple dataflows, the client should manually simulate caching of intermediate results by writing them to disc. Flink, however, has a dedicated `iterate` operator that can be used to express certain classes of iterative dataflows.

As a running example, consider again the `edges` and `paths` code snippet from Section 7.3. The *Emma Source* and *Emma Core* representations are depicted in Figure 7.2. The *Emma Core* expression matches the following criteria:

```
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)
```

Figure 7.2: *Emma Source* (left) and *Emma Core* (right) versions of a code snippet that can be specialized to Flink's `iterate` operator. For readability, not all code fragments in the *Emma Core* representation are translated to ANF form (indicated with a surrounding `ANF{...}` call).

- `k1` − `k3` form a control-flow graph corresponding to a simple `while` loop;
- `k1` has two parameters – an induction variable `i` of type `Int` and a `Bag` instance `paths`;
- the induction variable `i` binds to values in the $[0, N)$ range (in the example we have $N = 5$);
- except the induction variable update `i$2`, all value definitions in the loop body continuation `k3` form a dataflow graph rooted at `p$2`;
- `p$2` binds to the `Bag` parameter `paths` in the recursive `k1` call.

Because of that, we can replace the `k1` − `k3` loop with a Flink `iterate` call. To that end, we eliminate the `k1` subtree as well as preceding values contributing only to the induction variable `i` (e.g., `it` and `i$1`). The rewrite

- wraps the original body (minus the induction update term) in a fresh lambda function `f$1`;
- re-defines `paths` as a value definition that binds to the result of a `FlinkNtv.iterate` call;
- appends the body of the original suffix `k2` to the enclosing root continuation.

In our running example, the resulting expression 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)(p$1)
... // suffix
```

The `iterate` primitive is defined in a `FlinkNtv` module and delegates to Flink's native `iterate` operator. From our discussion in Example 2.4, recall that because Flink's `iterate` virtualizes the notion of an iterative dataflow, Flink's optimizer can analyze the body and automatically cache loop-invariant data. In order to avoid naïve caching of loop-invariant `Bag` instances, SPECIALIZE-LOOPS precedes ADD-CACHE-CALLS in the Flink-specific OPTIMIZE chain.

8 Implementation

The translation from *Emma Source* to *Emma Core* presented in Chapter 6 and the optimizations discussed in Chapter 7 were prototyped in Scala. In this section, we discuss some critical decisions and techniques employed in our prototype implementation. Section 8.1 outlines the core set of design principles guiding our decisions. Based on those, in Section 8.2 we discuss the trade-offs of possible metaprogramming infrastructures and explain the choice of Scala's macro and reflection-based APIs as the underlying foundation for our prototype. Finally, in Section 8.3 through Section 8.5 we discuss a number of implementation strategies used to overcome the challenges outlined in Section 8.2.

8.1 Design Principles

The main design principles guiding our implementation align with the general eDSL design objectives outlined in Section 2.2 – reuse as much Scala syntax as possible and at the same time minimize the number of idiosyncratic patterns required to encode DSL terms in Scala. In addition, to position *Emma* as a lightweight alternative to Spark's `RDD` and Flink's `DataSet` APIs, we aimed for an implementation that integrates well with off-the-shelf versions of Spark and Flink and does not require custom builds of Flink, Spark, or Scala. Finally, while the optimizing rewrites presented in Chapter 7 are data-independent, data-dependent optimizations were anticipated as part of future research. To facilitate both kinds of optimizations, we wanted our *Emma* compiler capable of staging, transforming, and compiling DSL terms both at compile- and at run-time.

8.2 Design Space

Development of the *Emma* prototype commenced in early 2014. At that time, the Scala ecosystem offered two different platforms for implementing optimizing DSLs – Scala

Macros and Lightweight Modular Staging (LMS). In Section 8.2.1 and Section 8.2.2, we discuss the benefits and drawbacks of these platforms, motivating the choice of Scala Macros in view of the design objectives outlined above. In Section 8.2.3, we mention other platforms and tools that have recently emerged, discussing their suitability for eDSL designs similar to the one proposed in this thesis.

8.2.1 LMS

LMS [RO10, RO12] is a framework for rapid development of embedded DSLs based on the concepts of *staging* [JS86, TS00] and *partial evaluation* [JGS93]. DSL programs are staged to an intermediate representation and optimized by means of partial evaluation in a series of successive stages. Each stage evaluates a staged program into a new program representation to be consumed by the next stage. Finally, the resulting program representation is translated into executable code. To illustrate the idea of modular staging advocated by LMS, we use the `power` function example from [RO10]. The `power` definition can be made available in objects and classes inheriting from the `Power` trait[1].

```
trait Power {
  def power(b: Double, x: Int): Double =
    if (x == 0) 1.0 else b * power(b, x - 1)
}
```

In LMS, staged terms are delimited via type-based annotations – the type `Rep[T]` denotes a staged computation whose unstaged variant will yield a value of type `T`. Unstaged expressions are partially evaluated in the current stage. In the above code snippet, we want to stage the parameter `b` and the return type of the `power` function. To achieve that we simply change their type from `Double` to `Rep[Double]`. The resulting version partially evaluates the exponent `x` – that is, the resulting `Rep[Double]` program represents a `power` computation specialized for a specific value of `x`.

```
trait Power {
  def power(b: Rep[Double], x: Int): Rep[Double] =
    if (x == 0) 1.0 else b * power(b, x - 1)
}
```

This modified Scala code snippet will not compile initially because the compiler will not be able to find (*i*) an implicit staging of the `Double` literal `1.0` to a `Rep[Double]` value, and (*ii*) a staged variant of the $*$ method which operates on `Rep[Double]` instead of on plain `Double` types. The approach advocated by LMS is to bundle and install such operations in a modular manner, using the so-called *cake pattern* [Hun13]. To make the example above compile, we must constrain the `this` type of the enclosing `Power` trait.

[1] In Scala, objects and classes can inherit from multiple traits.

```
trait Power { this: Arith =>
  def power(b: Rep[Double], x: Int): Rep[Double] =
    if (x == 0) 1.0 else b * power(b, x - 1)
}
```

The `this : Arith` type constraint asserts that objects and classes inheriting from `Power` also inherit from `Arith`, which provides the staged versions of $*$ and `1.0`. Other modules of staged functions are provided by different traits in a similar manner. For example, if a staged program relies both on arithmetic operations and on trigonometric functions such as `sin` and `cos`, the enclosing trait needs to be constrained with `Arith` and `Trig`.

```
trait SomeTrait { this: Arith with Trig =>
  ... // access to staged versions of *, +, cos, and sin
}
```

The `LMS` package provides default implementations for all modules. The method definitions in these implementations simply construct an ANF representation of the staged program term. For example, a `power(b, 3)` call in the first stage will partially evaluate the recursive calls `power` based on the (unstaged) `x` parameter. The result, consumed by the second stage, will be a `Rep[Double]` value representing the following ANF program.

```
val x0 = 1.0
val x1 = b * x1
val x2 = b * x2
val x3 = b * x3
x3
```

As part of the staging process, the framework also implicitly performs Common Subexpression Elimination (CSE), ensuring that the resulting ANF representation does not contain duplicated code.

Code generation in LMS is done explicitly with a dedicated `compile` call provided by a `Compile` trait. Depending on the used `Compile` implementation, the framework can generate code for different backends, e.g. Scala or C. For example, the following definition allows for instantiating specialized `power`$(\cdot,$ `x`$)$ implementations via `fastpower(x)` calls.

```
object fastpower extends Power with CompileScala {
  def apply(x: Int): Double => Double = compile {
    (b: Rep[Double]) => power(b, x)
  }
}
```

An optimizing DSL for parallel collection processing which follows the design outlined in Chapter 5 through Chapter 7 of this thesis can be realized on top of the LMS framework using the following implementation guidelines.

(**G1**) Define a `BagOps` trait which provides staged versions of the `Bag`A and `BagCompanion` API from Figure 5.2.
(**G2**) Implement the LIFT transformation from Chapter 6. Since the ANF and DSCF conversion is already handled by the staging facilities provided by LMS, we only have to implement NORMALIZE$_{\texttt{Bag}}$ and RESUGAR$_{\texttt{Bag}}$.
(**G3**) Implement the optimizing transformations from Chapter 7.
(**G4**) Implement backends specializing staged `Bag` operators to either `SparkBag` or `FlinkBag` operators and use them in conjunction with the `CompileScala` backend.
(**G5**) Define `onSpark` and `onFlink` compilation pipelines using chains of the above stages.

Realizing *Emma* on top of LMS, however, is problematic with respect to some of the objectives identified in Section 8.1. In the following, we discuss some of these problems.

First, while staging based on type annotations offers fine-grained control over which paths of the original program are staged, it also imposes a higher technical barrier for the eDSL users. In our running example, understanding the concepts of staging and partial evaluation was required in order to decide which types of the original `power` function should be adapted from `T` to `Rep[T]`. One simple way to eliminate this complexity dimension is to always stage the entire program. In type-based staging, this means changing the type of all terms from `T` to `Rep[T]`. However, this approach introduces some level of linguistic noise and violates the linguistic reuse principle from Section 2.2. With quotation-based embedding, the same effect is achieved by a single quotation and thereby requires fewer changes to the concrete syntax of the original program.

Second, the *Emma Source* language defined in Figure 5.1 assumes an open universe of methods and modules. This assumption is important for predictive analytics applications, as those typically use logic provided by third-party libraries. In the data integration and preprocessing phase, the elements of the input datasets are often normalized and vectorized using domain-specific methods such as Gaussian curve fitting or Radial Distribution Function (RDF) conversion. Data practitioners often rely on libraries that provide trusted implementations of these methods. In the Flink and Spark APIs, vectorization and normalization methods provided by third-party libraries can be easily called in lambdas passed to higher-order functions such as `map` or `reduce`. The LMS staging scheme outlined above, however, does not offer a mechanism to stage an open universe of methods and symbols. Therefore, in an LMS-based implementation of *Emma*, DSL users would have to extend the compilation infrastructure in an ad-hoc manner in order to add staging and code generation support for all library methods used in *Emma* pipelines. As before, we would like to remove compiler and code generation aspects from the user-facing API. An implementation based on Scala macros and quotations offers a straight-forward solution to the problem, as an open universe of methods is directly supported in the AST of the quoted terms.

Third, the LMS framework requires a modified version of the Scala runtime called Scala-Virtualized [RAM+12]. This requirement is dictated by the need to employ the method-based staging strategy outlined above to language features such as variable declarations and assignments, control-flow and pattern matching statements, and record types. To achieve that, the Scala runtime is modified in order to represent these features as virtual method calls. The semantics of these methods then can be overloaded by hosted DSLs or DSL frameworks such as LMS. Unfortunately, this modification is at odds with the requirement to integrate *Emma* with off-the-shelf versions of Flink and Spark. As both frameworks depend on Scala, an LMS-based implementation of *Emma* will only work with modified versions of Flink and Spark which are based on Scala-Virtualized. Again, a macro-based implementation is not affected by this problem – Scala macros ship as experimental feature with vanilla Scala since version 2.10 and are therefore compatible with any vanilla Flink or Spark distribution.

8.2.2 Scala Macros and Scala Reflection

Starting from version 2.10, Scala ships with experimental metaprogramming support consisting of two separate libraries. Scala macros [Bur13] offer facilities for compile-time metaprogramming, while Scala reflection [COD08] provides runtime reflection support. An important aspect is that the two libraries are based on the same API and share a substantial amount of code.

A Scala `def` method can be declared as a macro as follows[2].

```
def assert(cond: Boolean, msg: Any): Unit = macro Asserts.assertImpl
```

The signature of the macro implementation method `Asserts.assertImpl` mirrors the signature of `assert`.

```
def assertImpl(cond: c.Expr[Boolean], msg: c.Expr[Any]): c.Expr[Unit] = ...
```

In the above definition, `c` is a variable containing the enclosing macro `Context`, and the path-dependent type `c.Expr[T]` wraps an AST of type `T`. Client calls of the `assert` method are delegated to the `assertImpl` macro at compile-time using the ASTs of the passed arguments. The macro returns the AST of a program of type `Unit` wrapped in a container of type `c.Expr[Unit]`. The resulting expressions are inlined at the `assert` call sites. For example, the call

```
assert(x < 10, "limit exceeded")
```

[2] The example is adapted from the official Scala documentation

will result in an `assertImpl` call where the `c.Expr` parameters wrap the following ASTs.

```
// AST for the ‘cond‘ argument (x < 10)
Apply(
  Select(Ident(TermName("x")), TermName("$less"),
  List(Literal(Constant(10)))))

// AST for the ‘msg‘ argument ("limit exceeded")
Literal(Constant("limit exceeded"))
```

The `assertImpl` implementation can inspect the structure of these ASTs and use it to generate its output. For example, if the `cond` argument is an AST corresponding ot the `false` literal, it can return an expression node that simply wraps the `Unit` value.

```
// AST for the ‘cond‘ argument
Literal(Constant(false))

// AST for the result expression
Literal(Constant(Unit))
```

As outlined in Section 8.2.1, Scala macros and Scala reflection do not suffer from the problems associated with the LMS-based approach. First, a quotation-based design based on Scala macros allows for deep reuse of native Scala syntax with minimum amount of linguistic noise. To achieve that, we simply define `onSpark` and `onFlink` as polymorphic macros that can access the `AST` of their enclosing expression. Second, the metaprogramming API can access Scala's internal symbol table. In the above example, the `Select` node of the `cond` parameter has a `symbol` field which points to the unique '<' method symbol, and the `Ident` node has a `symbol` field which points to the unique term symbol associated with `x`. An implementation based on Scala macros therefore can easily incorporate an open universe of methods and types. Third, Scala macros and runtime reflection can be used out of the box with the latest versions of Flink and Spark. In addition, similar to LMS, Scala's reflection API ships with a lot of useful tooling and infrastructure, e.g. for tree traversal and transformation, manipulation of symbols and types, and AST inspection. Again, an implementation of *Emma* based on Scala macros can reuse this functionality.

Despite the benefits stated above, Scala macros and Scala reflection also exhibit a number of deficiencies when considered as foundation for the *Emma* DSL. First, there is a mismatch between the tree structure of Scala AST terms and the abstract syntax of *Emma Source* (Figure 5.1) and *Emma Core* (Figure 6.1 and Figure 6.4). To illustrate this, consider again the AST for the `x < 10` code fragment depicted above. An *Emma Core* tree depicting this term would consist of a single `DefCall` node.

```
DefCall(
```

```
Some(TermRef(x)),
/* method symbol for '<' */,
Seq.empty[Type], /* no type arguments */
Seq(Seq(Literal(10)))) /* A single singleton parameter list */
```

A solution to the AST mismatch problem is outlined in Section 8.3.

Second, the tree traversal and manipulation logic provided by Scala's metaprogramming API is too rudimentary and lacks support for commonly used high-level code manipulation and inspection patterns. As an example, consider a utility method that associates each subtree with its set of referenced binding symbols (i.e., the $\mathcal{R}$ function used in the DSCF and COMBINE transformations from Figure 6.3 and Figure 7.1). A high-level compiler API overcoming these limitations is presented in Section 8.4.

Third, Scala macros and Scala reflection share structurally identical, yet incompatible metaprogramming APIs. This is a consequence of the fact that the API types, operations and fields are imported in a path-dependent way through a dedicated `Universe` instance. At compile time, the enclosing universe can be accessed through the macro `Context` (`c.universe`), while at runtime it is available statically through `scala.reflect.runtime.universe`. Because of this, it is challenging to ensure that DSL compiler code (*a*) can be shared between compile-time and runtime components and (*b*) is organized in a modular manner. Section 8.5 discusses how our prototype compiler code is organized in view of this objective.

8.2.3 Current Solutions

Various solutions proposing different improvements over state-of-the-art tooling for staged compilation and metaprogramming in the Scala ecosystem have emerged after the inception *Emma*. Here, we briefly discuss those that might be a useful foundation for future implementations of the ideas presented in this work.

Scalamacros

Scalamacros[3] is a metaprogramming library which has been influenced by experiences and lessons learned in developing the `scala.reflect`-based macro system and its successor Scalameta[4]. The development roadmap for Scalamacros positions them as the long-term, production-ready successor of the experimental `scala.reflect`-based macros currently shipped with Scala. The main benefit of Scalamacros is a novel design approach where macros operate on a portable syntax abstraction decoupled from the AST of the underlying Scala compiler [LB17]. This leads to better tooling support, deeper IDE integration,

[3] http://github.com/scalacenter/macros
[4] http://scalameta.org

and painless migration of existing macros to new versions of Scala. Although it was developed independently from the Scalamacros effort, the encoding technique presented in Section 8.3 is quite similar to the approach proposed by Liu and Burmako [LB17]. An implementation of *Emma* on top of Scalamacros is likely to benefit from this similarity.

Squid

Another metaprogramming framework that has been recently proposed is *Squid* [PVSK18]. Squid combines the flexibility of dynamic quasi-quotes (in the style pioneered by Lisp) with the typing and scoping guarantees of static quasi-quotes (in the style pioneered by MetaML [TS00]). Squid can be used as an LMS alternative using a technique called *quoted staged rewriting* [PSK17]. A Squid-based implementation of *Emma* therefore will reconcile the simplicity of quotation-based delimiting of DSL terms with the elegance and power of staging as a principle method for program optimization.

Fusion-Enabling Transformation API

Matryoshka is a library that provides generalized folds, unfolds, and traversals for fixed-point data structures in Scala. The functionality offered by Matryoshka overlaps with recent work in structured recursion schemes by Hinze and Wu [HWG13, HW16]. Because the supported recursion schemes satisfy algebraic properties such as the Banana-Split and Cata-Fusion laws from Chapter 4, *Matryoshka*-based tree manipulation APIs automatically support a number of fusion-based optimizations, leveraging the construction of *nanopass compilers*. This allows to reconcile the software engineering benefits of structuring code around semantically isolated tree transformers with the performance benefits of executing a fused version of the chain of transformers constituting the DSL compiler.

The formal approach adopted by Matryoshka, however, also imposes a higher technical barrier for compiler developers, as they need to understand concepts such as catamorphism, anamorphism, zygomorphism, etc in order to use the Matryoshka API. Mapping tree traversals conceptualized as a set of inference rules to the right recursion scheme could be a challenging task, especially for people with no prior experience. To that end, Petrashko et al. [PLO17] offer a more pragmatic approach called *miniphases*. While Matryoshka advocates fusion based on soundness criteria inherent from the mathematical theory behind the underlying recursion schemes, the miniphases approach advocates for fusion based on high-level criteria decided by the developer. Compiler developers provide a list of tree invariants that each tree transformation is guaranteed to satisfy, and the compiler automatically checks these invariants during execution. Extensive testing is identified as a principle method to identify and mitigate errors in fused transformations.

Either of these two approaches will allow to encode the transformations presented

in Chapter 5 through Chapter 7 in a modular way and at the same time construct fast versions of the `onFlink` and `onSpark` compilation pipelines due to the applied transformation fusion.

8.3 Object Language Encoding

A practical problem that occurred when we implemented *Emma* on top of the Scala macros infrastructure was the mismatch between the AST representation of macro-based Scala terms and the abstract syntax of the object languages defined in Figure 5.1, Figure 6.1 and Figure 6.4. The main cause for this mismatch was our desire to decouple the abstract syntax forms used by *Emma* from the specifics of Scala's AST encoding in order to simplify the definitions of *Emma Core* and *Emma Source*. In the following, we give a couple of examples that illustrate why this simplification was desirable.

As discussed in Section 8.2.2, the Scala macros API exposes the same AST data structure as the one used internally by the Scala compiler. Consequently, the shape of the ASTs reflects some of the inner workings of the Scala compiler. To illustrate this, consider the following two code snippets.

```
var i = 0
var r = 0
do {
  i = i + 1
  r = r * i
} while (i < x)
```

```
var r = 0
var i = 0
while (i < x) {
  i = i + 1
  r = r * i
}
```

Internally, the loops in the above code snippets are represented by a `LabelDef` node and a nested `If` node. The condition and branches of the `If` node as well as the body of the `LabelDef` node are derived from the original loop condition and body. The shape of the actual ASTs is represented by the following Scala-like code.

```
doWhile$1() { // label definition
  { // loop body
    i = i + 1
    r = r * i
    ()
  }
  if (i < x) // loop condition
    doWhile$1() // label call
  else ()
}
```

```
while$1() { // label definition
  if (i < x) { // loop condition
    { // loop body
      i = i + 1
      r = r * i
      ()
    }
    while$1() // label call
  } else ()
}
```

As part of the code generation phase, the Scala compiler converts label calls to Java Virtual Machine (JVM) jump bytecode instructions. The *Emma Source* language depicted on Figure 5.1, however, is based on first-class `while` and `do − while` syntax.

Another source of syntactic diversity in macro ASTs stems from the variety of supported method calls, as illustrated by the following lines.

```
bar(1, 2) // monomorphic, unqualified
Foo.bar(1, 2) // monomorphic, qualified
baz[Int](1, 2) // polymorphic, unqualified
Foo.baz[Int](1, 2) // polymorphic, qualified
```

The corresponding ASTs of these method calls look as follows.

```
Apply(Ident(/*bar*/), /*(1,2)*/)
Apply(Select(Ident(/*Foo*/), /*bar*/), /*(1,2)*/)
Apply(TypeApply(Ident(/*baz*/), /*Int*/), /*(1,2)*/)
Apply(TypeApply(Select(Ident(/*Foo*/), /*baz*/), /*Int*/), /*(1,2)*/)
```

In each of the above examples, the top-level AST node is `Apply` and the argument list used as its second child is identical. The child representing the applied method, however, differs based on the method type and the shape of the application. For monomorphic methods, this can be either an `Ident` node denoting an unqualified method declared on an enclosing instance, or a `Select` node denoting the selection path of a qualified method. Orthogonally, the method denotation of polymorphic methods is wrapped in a `TypeApply` node which denotes application of type arguments.

In order to simplify the definition and reasoning of program transformations, our goal was to remove syntactic diversity illustrated above in the abstract syntax of the developed eDSL. At the same time, we wanted to remain compatible with the macro AST in order to reuse the macro API whenever possible. As a pragmatic solution, the syntactic forms outlined in Figure 5.1, Figure 6.1 and Figure 6.4 are encoded as *virtual nodes*. A virtual node is an object which defines a pair of `apply` and `unapply` methods which respectively construct and deconstruct a macro AST. For example, the virtual node corresponding to the `while` syntax in Figure 5.1 has the following form.

```
object While extends Node {
  def apply(cond: Tree, body: Tree): LabelDef = ...
  def unapply(loop: LabelDef): Option[(Tree, Tree)] = ...
}
```

This enables convenient Scala syntax for construction and deconstruction of `While` loops.

```
val loop = While(cond, body) // construct a While loop
loop match { case While(cond, body) => ... } // deconstruct a While loop
```

Note that the arguments and the return types of the `apply` and `unapply` functions in the `While` object are of type `Tree`, ensuring that we operate on macro AST values. This

allows to seamlessly integrate the Scala macro API in the *Emma* compiler. For example, we can retrieve the `Type` of the `cond` AST (which should be `Boolean`) using `cond.tpe`.

An important aspect of this implementation strategy is the ability to encode DSL syntax which does not have a natural mapping to macro AST fragments. A good example is the first-class comprehension syntax of *Emma Core* (see Figure 6.4). Since Scala desugars `for`-expressions as part of the parsing phase, the corresponding syntax is not available in the macro AST representation. To define a virtual node, in such situations we rely on auxiliary dummy methods. In the case of `for`-comprehensions, the dummy interface looks as follows.

```
object ComprehensionSyntax {
  def generator[A, M[_]](in: M[A]): A = ???
  def comprehension[A, M[_]](block: A): M[A] = ???
  def guard(expr: Boolean): Nothing = ???
  def head[A](expr: A): A = ???
}
```

Emma Core syntax can be mapped to Scala source code fragments utilizing the dummy methods listed above. For example, the *Emma Core* `for`-comprehension

$$\texttt{for}\ \{\ \texttt{x} \leftarrow \{\ xs\ \}\ ;\ \texttt{y} \leftarrow \{\ ys\ \}\ \}\ \texttt{yield}\ \{\ \texttt{val z} = (\texttt{x}, \texttt{y});\ \texttt{z}\ \}$$

is encoded by the following Scala source code fragment.

```
comprehension[(Int, Int), Bag] {
  val x = generator[(Int, Int), Bag] { xs }
  val y = generator[(Int, Int), Bag] { xs }
  head {
    val z = (x, y)
    z
  }
}
```

The encoding allows to define the `apply` and `unapply` methods of the virtual nodes corresponding to the syntactic forms of *Emma Core* `for`-comprehensions. For example, the `Generator` node for the $x \leftarrow \{\ xs\ \}$ generator will construct and match the macro AST corresponding to the second line of the above Scala encoding.

8.4 Tree Manipulation API

We provide a fluid functional API for *trans*forming and tra*versing* (shortened as *transversing*) Scala ASTs inspired by the Traversal Query Language (TQL) which has been recently proposed for Scalameta [BB15]. The API was designed with the following goals. First,

avoid explicit recursion by decoupling the matching rules from the transversal strategy. While the rules are always specific to the concrete transversal, transversal strategies can be abstracted as a finite set of available options supported by the API. Second, avoid use of mutable state. Instead, provide infrastructure for deriving tree attributes and an API to expose those to the matching rules during transversal.

8.4.1 Strategies

The core of the transversal API is built on top of the strategies described in [vdBKV03]. A transversal strategy is uniquely determined as a point in a two-dimensional space.

The first dimension determines the order in which nodes are visited. With a *top-down* strategy parents are visited before their children. Conversely, with a *bottom-up* strategy children are visited before their parents.

When a node is visited, the transversal strategy attempts to match it against one of the available rules. The second dimension determines the continuation criteria in the case of a match. The *continue* strategy continues with the next node in the selected order. The *break* strategy stops the transversal process after the first rule match. Finally, the *exhaust* strategy recursively applies all matching rules at a given node and then continues to the next node in the selected order.

For example, the strategy for the ANF transformation from Figure 6.2 is (*bottom-up*, *continue*), the DSCF transformation from Figure 6.3 uses (*top-down*, *continue*), and the NORMALIZE$_{\mathsf{M}}$ transformation from Figure 6.6 uses (*bottom-up*, *exhaust*).

The API offers fluent syntax for transversal construction. For example, the definition of the ANF transformation has the following shape (code snippets for attribute and rule declarations are given in the next sections).

```
val anf = api
  .TopDown.continue
  // zero or more attribute declarations
  // rule declaration
```

8.4.2 Attributes

All transversal strategies can operate on attributed trees. Declared attributes are attached to each node in the tree and can be made available to the matching rules during transversal. Depending on the derivation strategy, attributes can be synthesized, inherited, or accumulated. In each of these cases, the attribute is defined in terms of a partial function $a : \mathit{Tree} \to A$ and a monoid $M = (A, \odot, 1)$ with carrier coinciding with the attribute type A.

Inherited attributes are derived in a top-down manner along the recursion path of the transversed tree. Let t_i, $1 \leqslant i \leqslant n$ be the current path from the root of the tree t_1 to the currently visited node t_n. Set $x_i = a(t_i)$ if a is defined at t_i or $x_i = 1$ otherwise. The value of the inherited attribute at node t_n is defined by the following equation.

$$\textsc{inh}_a^M[\![t_n]\!] = x_1 \odot \ldots \odot x_{n-1} \qquad \text{(Attr-Inh)}$$

For example, consider $a : \mathit{Tree} \to \mathit{Option}[\mathit{Tree}]$ to be the (total) function $t \mapsto \mathit{Some}(t)$. If $1 = \mathit{None}$ and the $\odot$ rule selects the left-most element of the evaluated term, $\textsc{inh}_a^M[\![t_n]\!]$ wraps the root of the traversed tree and is None if and only if t_n is the root. Conversely, if the $\odot$ rule selects the right-most element, $\textsc{inh}_a^M[\![t_n]\!]$ denotes the parent of t_n and is None if and only if t_n is the root. To illustrate the associated API, consider the following code which declares an inherited attribute collecting all ancestors of the current node (the vector concatenation monoid is passed as implicit argument and is not shown).

```
.inherit(Attr.collect[Vector, Tree] {
  case ancestor => ancestor
})
```

Synthesized attributes are derived in a bottom-up manner from the current subtree. A synthesized attribute is conceptually similar to a catamorphism. Let t_n be the current node and t_i, $1 \leqslant i < n$ be its children. As before, set $x_i = a(t_i)$ if a is defined at t_i or $x_i = 1$ otherwise. The value of the synthesized attribute at node t_n is defined by the following equation.

$$\textsc{syn}_a^M[\![t_n]\!] = x_1 \odot \ldots \odot x_n \qquad \text{(Attr-Syn)}$$

Synthesized attributes are often maps of key-value pairs. The associated monoid operation merges two maps in a suitable way, e.g. by summing up values with the same key. For example, the following code snippet declares a synthesized attribute which counts the number of assignments for each variable in the associated subtree (as above, the monoid is passed implicitly to the `synthesize` function call).

```
.synthesize(Attr.group {
  case VarMut(sym, _) => sym -> 1
})
```

Finally, *accumulated attributes* are derived along the visiting trace determined by the selected transversal strategy. Let t_i, $1 \leqslant i < n$ be the trace of nodes visited so far and t_n be the current node. Set $x_i = a(t_i)$ if a is defined at t_i or $x_i = 1$ otherwise. The value of the accumulated attribute at node t_n is defined by the following equation.

$$\textsc{acc}_a^M[\![t_n]\!] = x_1 \odot \ldots \odot x_n \qquad \text{(Attr-Acc)}$$

For example, in conjunction with a *top-down* strategy the following code snippet will keep track of all method parameters seen so far.

```
.accumulate { case DefDef(_, _, paramss, _) =>
  for (ParDef(sym, _) <- paramss.flatten) yield sym
}
```

The lists of parameter symbols emitted by the supplied partial function are concatenated by the default list monoid (passed implicitly to `accumulate` as with the examples above).

The attribute API is typed. The type of the transversal strategy is parametric, with type parameters A, I, and S denoting heterogenous lists of its accumulated, inherited, and synthesized attributes. This allows to expose declared attributes to the transversal rules in a type-safe manner.

8.4.3 Rules

Transversal rules are defined as a partial callback function and attached to a transversal declaration using a suitable method call. The canonical forms `traverse` and `transform` accept a callback function of type $\mathit{Tree} \to \mathit{Unit}$ and $\mathit{Tree} \to \mathit{Tree}$. Alternatively, the API also offers the forms `traverseWith` and `transformWith`. The argument type in these variants is changed from *Tree* to $\mathit{Attr}[A, I, S]$, where *Attr* is defined as follows.

```
case class Attr[A, I, S](tree: Tree, acc: A, inh: I, syn: S)
```

Callbacks used with `traverseWith` and `transformWith` therefore have access to the attributes associated with the matched tree nodes. In addition, the `Attr` object provides projections such as `Attr.inh` that select only one type of attributes along with the matched tree node. A syntactically complete example of a transformation based on the transversal API is shown in Figure 8.1.

8.5 Code Modularity and Testing Infrastructure

One of the key challenges of the macro-based implementation of *Emma* was to ensure that (*i*) code is organized in a modular manner and (*ii*) individual modules could be tested and integrated with off-the-shelf libraries and tools.

To achieve that, we made use of the fact that the macro-based and the reflection-based APIs implement the `Universe` trait and differ only in the path from which the `Universe` methods and types are imported (see Section 8.2.2). To abstract from the concrete API implementation, the *Emma* compiler structure is based on the cake pattern [Hun13]. At the top of the hierarchy is a trait which defines an abstract `Universe` member.

```
trait Common {
  val u: Universe
  ...
}
```

Emma compiler modules are defined as traits inheriting from `Common`. Smaller modules can be aggregated into bigger ones using intermediate traits. For example, the `Comprehension` trait aggregates the logic for re- and desugaring, normalization, and combination of comprehensions, and is therefore defined as follows.

```
private[compiler] trait Comprehension extends Common
  with Combination
  with Normalize
  with ReDeSugar {
  this: Core =>
  ...
}
```

The `this : Core` type constraint indicates that the `Comprehension` implementation depends on methods and types provided by `Core` module. At the top level, the modules are aggregated by a `Compiler` trait which has two implementations. The `MacroCompiler` is used as a base for the `onFlink` and `onSpark` macro definitions outlined in Section 6.5. The `RuntimeCompiler` is used for testing, as discussed below.

The `RuntimeCompiler` facilitates writing tests for specific transformations against snippets of code which are directly defined in the source code of the enclosing test class. The general layout of a test class looks as follows. First, construct a test pipeline and a reference pipeline using the API exposed by the `RuntimeCompiler` instance. Second, reify a code snippet representing the test input and pass the resulting Scala AST to the test pipeline. Third, reify a code snippet representing the expected output and pass the resulting Scala AST to the reference pipeline. Third, ensure that the results of the two pipelines are equal up to a renaming of the `val` and `var` definitions. As an example, consider the following case from the ANF test.

```
// actual AST
val act = anfPipeline(reify {
  15 * t._1
})

// expected AST
val exp = idPipeline(reify {
  val x$1: this.t.type = t
  val x$2 = x$1._1
  val x$3 = 15 * x$2
  x$3
})
```

```
// check for equality
act shouldBe alphaEqTo(exp)
```

This design provides a flexible foundation for future research based on the DSL representations discussed in this thesis. For example, the `RuntimeCompiler` can be used in conjunction with the `MacroCompiler` in order to explore data-dependent optimizations such as cost-based join-order estimation in the comprehension combination phase.

```
val anf = api.BottomUp
  // Prepend owner symbol to inherited attributes
  .withOwner
  // Prepend a Boolean flag which marks type trees
  // to inherited attributes
  .inherit { case tree => tree.isType }
  // Transform the attributed tree
  .transformWith {
    // Bypass type trees
    case Attr.inh(tree, true :: _) =>
      tree

    // Simplify receiver & arguments
    case Attr.inh(
      call @ src.DefCall(rcv, m, tps, argss), _ :: owner :: Nil) =>

      // Unnest subexpressions from receiver
      val (init, rcv1) = rcv match {
        case Some(src.Block(stats, expr)) =>
          (stats, Some(expr))
        case Some(_) =>
          (Seq.empty, rcv)
        case None =>
          (Seq.empty, None)
      }

      // Unnest subexpressions from arguments
      val stats = init ++ argss flatten flatMap {
        case src.Block(stats, _) => stats
        case _ => Seq.empty
      }

      val argss1 = argss map (_ map {
        case src.Block(_, arg) => arg
        case arg => arg
      })

      // Assign the final result to a fresh val
      val nme = api.TermName.fresh(m.name)
      val lhs = api.ValSym(owner, nme, call.tpe)
      val rhs = core.DefCall(rcv1, m, tps, argss1)
      val dfn = core.ValDef(lhs, rhs)
      val ref = core.ValRef(lhs)

      // Wrap modified code in a block and return
      src.Block(stats :+ dfn)(ref)
  }
```

Figure 8.1: A simplified transformation example that brings method calls to ANF form – subexpressions in the method receiver `rcv` and the argument terms `argss` are assigned to fresh `vals`.

9 Evaluation

To assess the benefits of the optimizations from Chapter 7 we designed and conducted a set of experiments which we present and discuss in this chapter.

We ran the experiments on a local 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. The machines were connected with a Cisco 2960S switch. As dataflow 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 this memory for its managed runtime. Input and output data were stored in an HDFS 2.7.1 instance running on the same set of nodes.

Each of the experiments discussed in Section 9.1 through Section 9.4 was executed five times. The associated bar charts in Figure 9.1 through Figure 9.4 indicate the median run and the error bars denote the second fastest and second slowest runs. The experiments discussed in Section 9.5 were executed three times and the bars in Figure 9.5 indicate the median run.

9.1 Effects of Fold-Group Fusion

The first experiment demonstrates the effects of the fold-group fusion (FGF) optimization presented in Section 7.2. To asses those, we executed one iteration of the *k-means* clustering algorithm [For65]. As input data, we used synthetic datasets consisting of points sampled from one of k multivariate Gaussian distributions. The data generator was parameterizable in the centroid distribution function and in the dimensionality of the generated points. In total, we ran four experiments, using both uniform and Zipf distribution on each of the two backends. In each experiment, we scaled the dimensionality of the data points from 10 to 40 in a geometric progression. For every dataset, we compared the runtime of two *Emma*-based implementations with FOLD-GROUP-FUSION

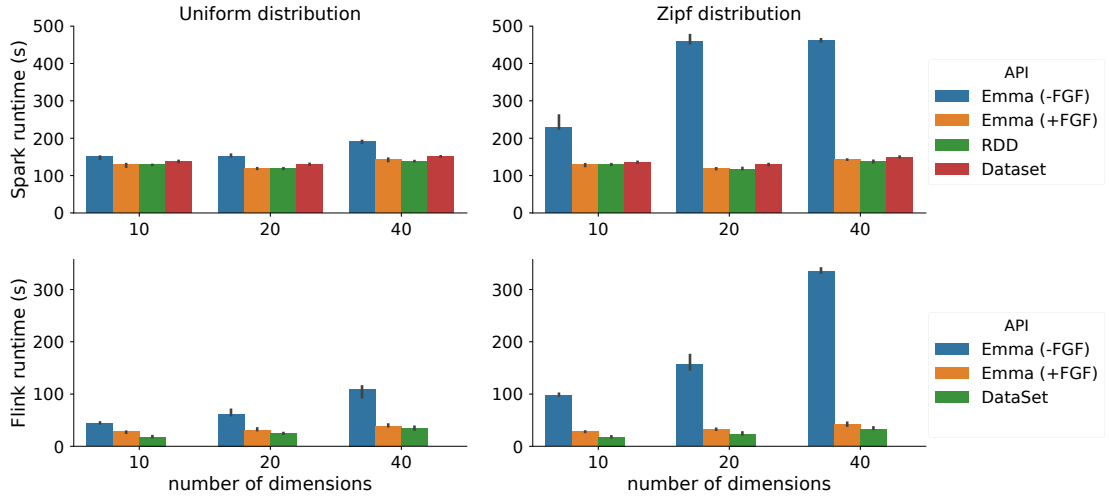


Figure 9.1: Effects of fold-group fusion (FGF) in Flink and Spark.

turned off (-FGF) and on (+FGF). As a baseline, we used a `DataSet` implementation for Flink and `Dataset` and `RDD` implementations for Spark.

The experiment results are presented in Figure 9.1. In the *Emma (-FGF)* version, the k means are computed naïvely with a `reduceByKey` ∘ `groupBy` operator chain in Flink and a `map` ∘ `groupBy` chain in Spark. Consequently, all points associated with a same centroid must be shuffled to a single machine where their mean is then computed. The total runtime therefore is determined by the size of the largest group. In contrast, when FGF is enabled, the sum and the 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. In the associated shuffle step we only need to transfer one partial result per group and per worker. The total runtime therefore does not depend on the group size. This effect is demonstrated by the experiment results. In both backends, the runtime of the *Emma (-FGF)* implementation grows as we increase the dimensionality of the data. For the *Emma (+FGF)* and the baseline variants, on the other hand, the runtime is not affected by the underlying centroid distribution and is only marginally influenced by changes in data dimensionality. The code generated *Emma (+FGF)* therefore performs on par with the code written directly against the Flink and the Spark APIs. The speedup of *Emma (+FGF)* with respect to *Emma (-FGF)* varies. In Flink, it ranges from 37% to 65% (Uniform) and from 72% to 88% (Zipf). In Spark, the ranges are from 14% to 26% (Uniform) and from 44% to 70% (Zipf). The effect grows stronger if the underlying centroid distribution is skewed, as this skew is reflected in the cardinality of the aggregated groups.

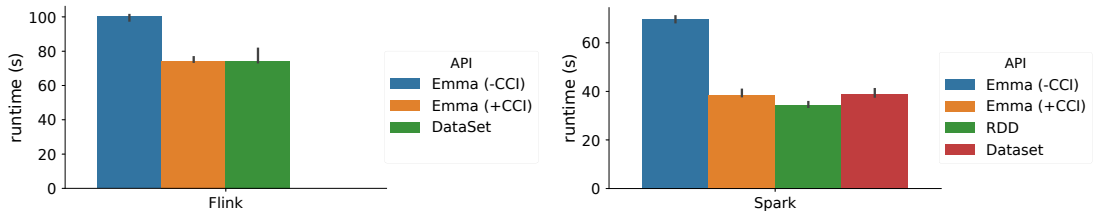


Figure 9.2: Effects of cache-call insertion (CCI) in Flink and Spark.

9.2 Effects of Cache-Call Insertion

The second experiment demonstrates the benefits of the cache-call insertion (CCI) optimization proposed in Section 7.3.

As input data, we used a snapshot of the Internet Movie Database (IMDb)[1] which was subsequently parsed and saved as structured collections of JSON objects. The workload for performs the following computations. In the first step, we perform a three-way join between movies, countries, and technical information, and select information about German titles categorized as "motion picture" which were released in the 1990s. In the second step, we filter six subsets of these titles based on different criteria (e.g., titles with aspect ratio 16:9 or titles shot on an Arri film camera) and collect the qualifying entries on the workload driver. In the *Emma (+CCI)* and the baseline variants, the collection obtained after the first step is cached, and in the *Emma (-CCI)* variant it is not.

The results are depicted on Figure 9.2. As in the previous experiment, the optimized *Emma* version is comparable with the baseline versions implemented directly on top of the backend APIs. Compared to the naïve version, the optimized variants achieve a speedup of 26% for Flink and 45% for Spark. The difference is due to the underlying caching mechanism. Spark has first-class support for caching and keeps cached collections directly in memory. Flink, on the other hand, does not support first-class caching. Consequently, the `FlinkOps.cache` primitive inserted by the *Emma* compiler is implemented naïvely by simply writing the cached distributed collection to HDFS. Subsequent reads of cached collections are therefore more expensive in Flink than in Spark. Nevertheless, the CCI optimization results in a significant improvement for both backends.

9.3 Effects of Relational Algebra Specialization

The next experiment investigates the benefits of relational algebra specialization (RAS) – specializing `map`, `withFilter`, and `join` calls in terms of the relational algebra operators `select`, `project`, and `join` provided by the Spark `Dataset` API (see Section 7.1.3).

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

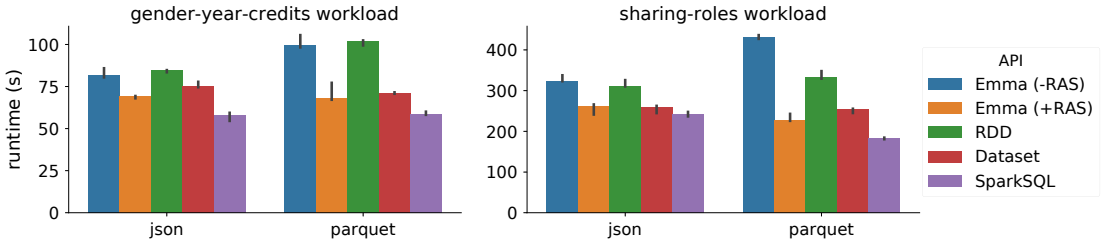


Figure 9.3: Effects of relational algebra specialization (RAS) in Spark.

As before, the experiments are based on the IMDb shapshot. To quantify the performance improvement of RAS we use two different workloads. The 'gender-year-credits' workload represents a simple three-way join where people and movies are connected via credits with credit type 'actor'. We emit 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 co-starred in a third movie. For example, Michael Caine (in "Sherlock Holmes, Without a Clue") and Roger Moore (in "Sherlock Holmes in New York") have both played Sherlock Holmes and acted together in "New York, Bullseye!". We include Spark SQL next to the `RDD` and `Dataset` baseline implementations as well as a more efficient columnar format (Parquet) next to the string-based JSON representation.

The results for the two workloads are depicted on Figure 9.3. In all four experiments, the *Emma (-RAS)* variant performs on par with the `RDD` implementation, and the optimized *Emma (+RAS)* variant is comparable with the `Dataset` implementation. Notably, the speedup for Parquet files (32% and 48%) is higher than the one for JSON (16% and 19%). The difference is explained by the more aggressive optimizations performed by Spark in the first case. `Dataset` dataflows which read data from Parquet can utilize Parquet's columnar format and push adjacent `select` and `project` operators directly to the Parquet reader. In *Emma*, as a result of the COMBINE translation scheme from Figure 7.1, local predicates are pushed directly on top of the base collections. A subsequent RAS therefore enables selection push-down performed by Spark. However, the current COMBINE scheme does not automatically insert projections. Consequently, in the Parquet experiments the compiled `for`-comprehensions in the *Emma (+RAS)* variants are respectively 15% and 20% slower than the Spark SQL implementation, which enables both selection and projection push-down. No narrow this gap the COMBINE translation scheme has to be augmented with a suitable projection rule.

9.4 Effects of Native Iteration Specialization

The last optimization which we investigate in isolation is the Flink-specific native iterations specialization (NIS) proposed in Section 7.4.

Like the CCI and RAS experiments, the NIS experiment is also based on the IMDb

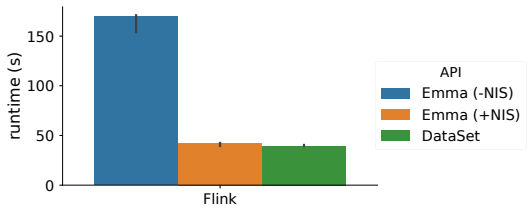


Figure 9.4: Effects of native iterations specialization (NIS) in Flink.

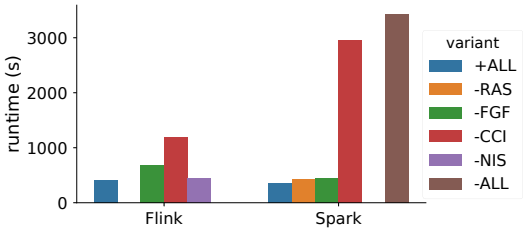


Figure 9.5: Cumulative optimization effects for the NOMAD use case.

snapshot. The workload first selects pairs of IDs identifying directors billed for the same movie for titles released between 1990 and 2010. The resulting relation is treated as a set of edges, and a subsequent iterative dataflow computes the first five steps of the connected components algorithm proposed by Ewen et al. in [ETKM12] (the variant we use is Fixpoint-CC from Table 1). 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 updates its own component ID to the minimum value of all received messages.

The *Emma (-NIS)* variant does not specialize the connected components loop as a Flink native iteration, but still performs the CCI optimization. The loop-independent collection of edges and the component assignments at the end of each iteration are consequently saved to HDFS by the inserted `FlinkOps.cache` calls. In the *Emma (+NIS)* variant, CCI is not needed as the Flink runtime manages the iteration state and loop-invariant dataflows in memory. Consequently, the *Emma (+NIS)* variant and the baseline `DataSet` implementation are 75% faster than the *Emma (-NIS)* variant.

9.5 Cumulative Effects

Finally, we investigate the cumulative effects of all optimizations using an end-to-end data analytics pipeline from a real-world use case.

The workload for this experiment is based on data obtained from the NOMAD repository[2]. The NOMAD repository contains a large archive of output data from computer simulations for material science in a common hierarchical format [GCL+16]. 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 files contain data about (1) the simulated physical systems and (2) the positions of the simulated atoms, as well as meta-information about (3) periodic dimensions and (4) simulation cells.

The workload pipeline looks as follows. In the first step, we join information from the four

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

CSV sources listed above and apply a Radial Distribution Function (RDF) conversion which yields a collection of dense vectors characterizing the result of each simulation. In the second step, we execute n runs of the first m iterations of a k-means clustering algorithm. We keep track of the optimal solution obtained at the end of each run and save it to HDFS at the end of the pipeline. To obtain sufficiently small numbers for a single experiment run, for the purposes of the presented experiment we choose $n = 2$, $m = 2$ and $k = 3$. In practice however, the values for n and m will likely be higher.

The workload is encoded as an *Emma* program and compiled in 5 different variants for each of the two supported backends. The *+ALL* (*-ALL*) variant denotes a compilation where all optimizations are enabled (disabled). The *-OPT* variant denotes a compilation where only the *OPT* optimization is disabled.

The results of the experiment are depicted on Figure 9.5. The Spark runtimes vary between 346s for the *+ALL* variant and 3421s for *-ALL*. In Flink, *+ALL* achieves 413s, while *-CCI* is slowest with 1186s (the *-ALL* variant did not finish successfully). For both scenarios, the largest penalty comes for a missing *CCI* optimization – 88% for Spark and 66% for Flink. With disabled *FGF*, the slowdown is 21% for Spark and 40% for Flink. Finally, omitting *RAS* results in 18% slowdown for Spark, and omitting *NIS* in 9% slowdown for Flink.

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

10 Related Work

This chapter reviews work related to the concepts and ideas presented in this thesis. Section 10.1 discusses work related to the mathematical foundations presented in Chapter 4. Section 10.2 discusses related DSLs.

10.1 Formal Foundations

The use of monads to structure and reason about computer programs dates back to Moggi [Mog91], who suggests them as a referentially transparent framework for modeling computations with effects. Comprehensions – a convenient, declarative syntax that can be defined in terms of a monad (essentially the $\mathcal{MC}$ scheme from Section 4.1.6) were introduced by Wadler [Wad92, Wad95]. Using comprehensions as a unifying foundation for database query languages for different bulk types (i.e. the types discussed in Section 4.1.4) can be traced back to the work of Trinder [TW89, DAW91, Tri91]. Notably, following unpublished work by Wadler [Wad90], Trinder suggests extending the monad with functions *zero* and *combine* to a structure called *ringad*. While the definition of Trinder requires only that *zero* is a unit of *combine*, adding associativity and commutativity yields the structure used as formal foundation for the API presented in Section 5.3.

Buneman and Tannen start from the basic notion of catamorphisms (i.e. structural recursion). They advocate that query languages should be constructed from the primitive notion of set catamorphisms [TBN91] and 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 [BNTW95]. These ideas are demonstrated by the Comprehension Language (CL) – a functional programming language for collection types based on comprehensions [BLS+94]. Notably, the IR proposed for CL does not make explicit use of a collection type monad – comprehension syntax in CL is defined directly in terms of catamorphisms on collections in union representation.

Similarly, Fegaras starts with the basic notion of monoids and proposes a core calculus

which defines comprehension syntax directly in terms of monoid catamorphisms [Feg94]. Fegaras and Mayer then show that the monoid calculus can be used to define the Object Query Language (OQL) – a standardized language for object-oriented DBMSs [FM95].

Despite some naming and notational differences, the formal development suggested in these two lines of work is quite similar. For example, the collection types associated with the *sr_comb* structural recursion scheme in [BNTW95] and the free monoids used in [Feg94] coincide. In addition, the catamorphic definitions of *ext* (Section 2.3 in [BNTW95]) and *hom* (Definition 5 in [Feg94]) both correspond to the higher-order function *flatmap*. Using the notation from Chapter 4, for a collection type T in union representation and a function $f : A \to B$ this definition looks as follows.

$$flatmap^{\mathsf{T}}(f) = (\!|emp_B^{\mathsf{T}} \triangledown f \triangledown uni_B^{\mathsf{T}}|\!)_A^{\mathsf{T}} \qquad \text{(Uni-Flatmap)}$$

The development closest to the exposition in Chapter 4 of this thesis is given by Grust [GS99, Gru99]. Similar to both Buneman and Fegaras, he starts from the basic notion of catamorphisms. Compared to the work discussed above, however, the work of Grust differs in the following aspects. First, he relies on collections in insert representation (although the union representation is discussed briefly in [Gru99]). Second, he explicitly derives a monad with zero from the associated algebra and uses it to define comprehension syntax using a translation scheme similar to the one suggested by Wadler. However, in contrast to the monad comprehension scheme from [Wad92], the one given by Grust supports generators ranging over multiple collection types, employing an implicit type coercion approach similar to the one proposed by Fegaras in [Feg94]. Third, Grust argues that comprehensions are a useful representation for defining and reasoning about optimizing program transformations. As such, he suggests that comprehensions should be part of the abstract syntax of an optimizing query compiler. Finally, he also suggests a compilation strategy based on rule-based translation of comprehensions using comprehension combinators.

The formal foundations used in this thesis follow Grust in all but the first aspect, where we opt for the union representation similar to Buneman and Fegaras. Our choice is motivated by the parallel nature of the underlying execution architectures. The intricate connection between Uni-Sign and its associated recursion scheme Uni-Fold for structuring parallel programs has already been highlighted by Skillicorn [Ski93a, Ski93b] and Steele [Jr.09]. Our contribution is in identifying the relevance of this methodology for the design of APIs and DSLs targeting parallel dataflow engines. In addition, extending a comprehension-based IR such as *Emma Core* with support for control-flow fills the semantic gap between previous work and typical use-cases for engines such as Spark or Flink.

Recently, Gibbons brought back attention to [Wad90] in a survey article [Gib16]. He argues that ringads and ringad comprehensions represent a better foundation and query

notation language than monads. Although we don't follow the ringad nomenclature, the work in this thesis obviously supports this claim. In addition, we highlight the connection between Uni-Asso and Uni-Comm in the ringad definition and data-parallel execution.

10.2 Related DSLs

DSLs related to *Emma* can be categorized in a two-dimensional space. The first dimension denotes the implementation strategy according to the classification scheme from Figure 2.1. The second dimension classifies DSLs based to their execution backend – a parallel dataflow engine, an RDBMS, or a custom runtime. In this section, we review related DSLs with Manhattan distance at most one – that is, stand-alone DSLs with a parallel dataflow backend and embedded DSLs with arbitrary backend. To the best of our knowledge, *Emma* is the first quotation-based eDSL that targets parallel dataflow engines.

10.2.1 sDSL Targeting Parallel Dataflow Engines

Pig [ORS+08] and Jaql [BEG+11] are stand-alone scripting DSLs that compile to a cascade of Hadoop MapReduce jobs. Hive [TSJ+09] provides warehousing capabilities on top of Hadoop or Spark using a SQL-like DSL, and SparkSQL [AXL+15] is a SQL layer on top of Spark developed as part of the Spark project. SCOPE [ZBW+12] is another SQL-like DSL developed by Microsoft which runs on a modified version of the Dryad dataflow engine [IBY+07]. Stand-alone DSLs such as the ones listed above provide automatic optimization (such as join order optimization and algorithm selection) at the cost of more limited expressive power. In particular, they lack first-class support for control flow and do not treat UDFs as first-class citizens. Optimizations related to these syntactic elements therefore are designed in an ad-hoc manner. For example, PeriSCOPE [GFC+12] optimizes SCOPE UDFs, but relies on ILSpy[1] for bytecode decompilation and Cecil[2] for code inspection and code synthesis. In contrast, the *Emma Core* IR presented in this thesis integrates both control flow and UDFs as first-class citizens. This enables definition and reasoning about optimizations related to these constructs in a unified methodological framework. At the same time, SQL-like optimizations can be integrated on top of the first-class comprehension syntax used in *Emma Core*.

10.2.2 eDSLs Targeting RDBMS Engines

The most popular example of an eDSL targeting RDBMS engines is Microsoft's LINQ [MBB06]. Database-Supported Haskell (DSH) [GGSW10] is an eDSL that facilitates database-supported execution of Haskell programs through the Ferry programming language [GMRS09]. As with stand-alone DSLs, the main difference between those

[1] http://wiki.sharpdevelop.net/ilspy.ashx
[2] http://www.mono-project.com/Cecil

languages and *Emma* is the scope of their syntax and IR. LINQ's syntax and IR are based on chaining of methods defined by an `IQueryable` interface. DSH is based on Haskell list comprehensions desugared by the method suggested by Jones and Wadler [JW07]. In particular, neither LINQ nor DSH lift control-flow constructs from the host language in their respective IRs. In addition, because they target RDBMS engines, these eDSLs 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 similarity between SQL-based eDSLs and *Emma* deserves further investigation. In particular, transferring avalanche-safety [GRS10, UG15] and normalization [CLW13] results obtained in this space to *Emma Core* is likely to further improve the runtime performance of compiled *Emma* programs.

10.2.3 eDSLs Targeting Parallel Dataflow Engines

The eDSLs exposed by the Spark and Flink systems and their problems are discussed in detail in Section 2.3. A number of similar system-independent eDSLs have been also proposed. FlumeJava [CRP+10] and Cascading[3] provide an abstraction API for dataflow graph assembly with pluggable dataflow engines and dedicated execution planner. Similarly, Summingbird [BROL14] and Apache Beam[4] (an open-source descendant of the Dataflow Model proposed by Google [ABC+15]) provide a unified API for stream and batch data processing which as well is decoupled from the execution backend. In all of the above examples, DSL terms are delimited based on their type. Consequently, they suffer from the deficiencies associated with the Flink and Spark eDSLs illustrated in Section 2.3.

Jet [AJRO12] is an LMS-based eDSL which supports multiple backends (e.g. Spark, Hadoop) and performs optimizations such as operator fusion and projection insertion. However, the Jet API is based on a distributed collection (`DColl`) which resembles more Spark's `RDD` than *Emma*'s `Bag` interface. In particular, the `DColl` relies on explicit `join` and `cache` operators and lacks optimizations which introduce those automatically.

The Data Intensive Query Language (DIQL) [FI17] is a SQL-like Scala eDSL. DIQL is based on monoids and monoid homomorphisms and therefore seems closest to the ideas presented in this thesis. A notable difference between DIQL and *Emma* is in their control-flow model. DIQL relies on a custom `repeat` construct, while *Emma* supports general-purpose `while` and `do − while` loops. In addition, DIQL's frontend is based on a custom string interpolator. Consequently, DIQL programs are specified as strings and therefore do not enjoy the linguistic reuse and tooling benefits of the quotation-based delimiting advocated by *Emma*.

[3] https://www.cascading.org/
[4] https://beam.apache.org/

10.2.4 eDSLs with Custom Runtimes

Delite [SBL+14] is a compiler framework for the development of data analytics eDSLs targeting parallel heterogeneous hardware. Delite's IR is based on functional primitives such as *zipWith*, *map* and *reduce*, and Delite eDSLs are defined in terms of these primitives. The framework compiles eDSLs programs to executable kernels using an LMS-like staging approach, and schedules these kernels with a purpose-built runtime. Implementing *Emma* on top of Delite requires (*a*) defining *Emma Core* in terms of Delite's IR and (*b*) adding support for Flink and Spark kernels to the Delite runtime. Since Delite is based on LMS, however, such an implementation will suffer from the issues outlined in Section 8.2.1.

Another Scala-based eDSL for unified data analytics is the AL language proposed by Luong et al. [LHL17]. AL programs are translated to a comprehensions-based IR and executed by a dedicated runtime employing just-in-time (JIT) compilation and parallel `for`-loop generation for the comprehensions found in the IR. Similar to AL, the *Emma* IR uses monad comprehensions as a starting point for compiler optimizations. However, *Emma Core* also emphasizes the importance of control-flow primitives that cannot be translated to comprehensions. In addition, similar to DIQL, ALs frontend is also based on a custom string interpolator and suffers from the same limitations.

11 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 identified and showcased a number of limitations shared between these eDSLs. A critical look of their design revealed that the common cause of these limitations is that DSL terms are delimited in the enclosing host language program based on their type. Consequently, IRs constructed from type-delimited eDSLs can only reflect host language method calls on these types. The declarativity and the optimization potential attained by type-delimited eDSLs are thereby heavily restricted.

As a solution, we proposed an eDSLs design where DSL terms are delimited using quasi-quotation. DSLs embedded in this manner can reuse more host language constructs in their concrete syntax and reflect those in their IR. As a result, quotation-based eDSLs can realize declarative syntax and optimizations traditionally associated with sDSLs such as SQL.

To support our claim, we proposed *Emma* – a quotation-based DSL embedded in Scala which targets Flink or Spark as co-processors for its distributed collection abstraction. We presented and discussed different aspects of the design and implementation of *Emma*. As a formal foundation, reflecting 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 and their realization using Scala's native `for`-comprehension syntax. As a basis for compilation, we proposed *Emma Core* – an IR which extends ANF with first-class comprehensions. To showcase the utility of *Emma Core* we developed a series of optimizations which solve the issues identified with state-of-the-art eDSLs in the beginning of the thesis. The performance impact of these optimizations for both backends was demonstrated with a range of optimization-specific experiments and an end-to-end data analytics pipeline.

The proposed design can be therefore seen as a first step towards reconciling the utility

of state-of-the-art eDSLs with the declarativity and optimization potential of sDSLs such as SQL. Nevertheless, in addition to collections, modern data analytics applications increasingly rely on data streams and tensors. In current and future work, we plan to extend the *Emma* API with types and APIs reflecting these abstractions. The primary goals thereby are twofold. First, ensure that different APIs can be composed and nested in an orthogonal manner. For example, a bag can be converted into a tensor (composition), or we can process a stream of tensors (nesting). Second, ensure that the degrees of freedom resulting from this orthogonality do not affect the performance of the compiled program. In particular, this entails designing and implementing optimizing program transformations that target DSL terms representing a mix of the available APIs.

Bibliography

[ABC+15] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[AJRO12] Stefan Ackermann, Vojin Jovanovic, Tiark Rompf, and Martin Odersky. Jet: An embedded dsl for high performance big data processing. In *International Workshop on End-to-end Management of Big Data (BigData 2012)*, number EPFL-CONF-181673, 2012.

[AKK+15] Alexander Alexandrov, Andreas Kunft, Asterios Katsifodimos, Felix Schüler, Lauritz Thamsen, Odej Kao, Tobias Herb, and Volker Markl. Implicit parallelism through deep language embedding. In *SIGMOD Conference*, pages 47–61, 2015.

[AKKM16] Alexander Alexandrov, Asterios Katsifodimos, Georgi Krastev, and Volker Markl. Implicit parallelism through deep language embedding. *SIGMOD Record*, 45(1):51–58, 2016.

[AKL+17] Alexander Alexandrov, Georgi Krastev, Bernd Louis, Andreas Salzmann, and Volker Markl. Emma in action: Deklarative datenflüsse für skalierbare datenanalyse. In Bernhard Mitschang, Daniela Nicklas, Frank Leymann, Harald Schöning, Melanie Herschel, Jens Teubner, Theo Härder, Oliver Kopp, and Matthias Wieland, editors, *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings*, volume P-265 of *LNI*, page 609, 2017.

[App98] Andrew W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[ASK+16] Alexander Alexandrov, Andreas Salzmann, Georgi Krastev, Asterios Katsifodimos, and Volker Markl. Emma in action: Declarative dataflows for

scalable data analysis. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 2073–2076, 2016.

[AXL+15] 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. Spark SQL: relational data processing in spark. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1383–1394, 2015.

[Bac88] Roland C Backhouse. *An exploration of the Bird-Meertens formalism.* 1988.

[BB15] Eric Béguet and Eugene Burmako. Traversal query language for scala. meta. Technical report, EPFL, 2015.

[BdM97] Richard S. Bird and Oege de Moor. *Algebra of programming.* Prentice Hall International series in computer science. 1997.

[BEG+11] Kevin Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh Carl-Christian Kanne, Fatma Ozcan, and Eugene J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 2011.

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

[Bir87] Richard S et al. Bird. An introduction to the theory of lists. *Logic of programming and calculi of discrete design*, 36:5–42, 1987.

[BLS+94] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension Syntax. *SIGMOD Record*, 1994.

[BNTW95] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.

[BROL14] P. Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. Summingbird: A framework for integrating batch and online mapreduce computations. *PVLDB*, 7(13):1441–1451, 2014.

[Bur13] Eugene Burmako. Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala, SCALA@ECOOP 2013, Montpellier, France, July 2, 2013*, pages 3:1–3:10, 2013.

[CB74] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A structured english query language. In Randall Rustin, editor, *Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, May 1-3, 1974, 2 Volumes*, pages 249–264, 1974.

[CLW13] James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 403–416, 2013.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[COD08] Yohann Coppel, Martin Odersky, and Gilles Dubochet. Reflecting scala. *Semester project report, Laboratory for Programming Methods. Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland*, 2008.

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

[DAW91] Phil Trinder David A. Watt. Towards a theory of bulk types, 1991.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[DS00] Olivier Danvy and Ulrik Pagh Schultz. Lambda-dropping: transforming recursive equations into programs with block structure. *Theor. Comput. Sci.*, 248(1-2):243–287, 2000.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations und Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. 1985.

[ETKM12] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning fast iterative data flows. *PVLDB*, 5(11):1268–1279, 2012.

[Feg94] Leonidas Fegaras. A uniform calculus for collection types. Technical report, Oregon Graduate Institute, 1994.

[FI17] Leonidas Fegaras and Ashiq Imran. Compile-time code generation for embedded data-intensive query languages. Under submission, July 2017.

[FLG12] Leonidas Fegaras, Chengkai Li, and Upa Gupta. An optimization framework for map-reduce queries. In Elke A. Rundensteiner, Volker Markl, Ioana Manolescu, Sihem Amer-Yahia, Felix Naumann, and Ismail Ari, editors, *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, pages 26–37, 2012.

[FM95] Leonidas Fegaras and David Maier. Towards an effective calculus for object query languages. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.*, pages 47–58, 1995.

[Fok92] Maarten M. Fokkinga. *Law and order in algorithmics.* 1992.

[Fok96] Maarten M. Fokkinga. Datatype laws without signatures. *Mathematical Structures in Computer Science*, 6(1):1–32, 1996.

[For65] E. Forgy. Cluster analysis of multivariate data: Efficiency versus interpretability of classification. *Biometrics*, 21(3):768–769, 1965.

[Fre87] Johann Christoph Freytag. A rule-based view of query optimization. In Umeshwar Dayal and Irving L. Traiger, editors, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 173–180, 1987.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In Robert Cartwright, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247, 1993.

[GCL+16] Luca M Ghiringhelli, Christian Carbogno, Sergey Levchenko, Fawzi Mohamed, Georg Huhs, Martin Lüders, Micael Oliveira, and Matthias Scheffler. Towards a common format for computational material science data. *arXiv preprint arXiv:1607.04738*, 2016.

[GFC+12] Zhenyu Guo, Xuepeng Fan, Rishan Chen, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Chang Liu, Wei Lin, Jingren Zhou, and Lidong Zhou. Spotting code optimizations in data-parallel pipelines through periscope. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 121–133, 2012.

[GGSW10] George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. Haskell boards the ferry - database-supported program execution for haskell. In Jurriaan Hage and Marco T. Morazán, editors, *Implementation and Application of Functional Languages - 22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers*, volume 6647 of *Lecture Notes in Computer Science*, pages 1–18, 2010.

[Gib94] Jeremy Gibbons. An introduction to the Bird-Meertens Formalism. Presented at 'New Zealand Formal Program Development Colloquium', Hamilton, November 1994, November 1994.

[Gib16] Jeremy Gibbons. Comprehending ringads - for phil wadler, on the occasion of his 60th birthday. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 132–151, 2016.

[GLSW93] Peter Gassner, Guy M. Lohman, K. Bernhard Schiefer, and Yun Wang. Query optimization in the IBM DB2 family. *IEEE Data Eng. Bull.*, 16(4):4–18, 1993.

[GMRS09] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. FERRY: database-supported program execution. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 1063–1066, 2009.

[GRS10] Torsten Grust, Jan Rittinger, and Tom Schreiber. Avalanche-safe LINQ compilation. *PVLDB*, 3(1):162–172, 2010.

[Gru99] Torsten Grust. *Comprehending Queries*. PhD thesis, Universität Konstanz, 1999.

[GS99] Torsten Grust and Marc H. Scholl. How to comprehend queries functionally. *J. Intell. Inf. Syst.*, 12(2-3):191–218, 1999.

[GW14] Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: deep and shallow embeddings (functional pearl). In *ICFP*, pages 339–347, 2014.

[Har13] Joseph J. Harjung. Reducing formal noise in pact programs. Master's thesis, TU Berlin, 2013.

[HPS+12] Fabian Hueske, Mathias Peters, Matthias Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. Opening the black boxes in data flow optimization. *PVLDB*, 5(11):1256–1267, 2012.

[Hun13] John Hunt. Cake pattern. In *Scala Design Patterns*, pages 115–119. 2013.

[HW16] Ralf Hinze and Nicolas Wu. Unifying structured recursion schemes - an extended study. *J. Funct. Program.*, 26:e1, 2016.

[HWG13] Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. Unifying structured recursion schemes. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 209–220, 2013.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In Paulo Ferreira, Thomas R. Gross, and Luís Veiga, editors, *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*, pages 59–72, 2007.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation.* Prentice Hall international series in computer science. 1993.

[Jr.09] Guy L. Steele Jr. Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful. In Graham Hutton and Andrew P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 1–2, 2009.

[JS86] Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*, pages 86–96, 1986.

[JW07] Simon L. Peyton Jones and Philip Wadler. Comprehensive comprehensions. In Gabriele Keller, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, pages 61–72, 2007.

[Kre15] Aljoscha Krettek. Using meta-programming to analyze and rewrite domain-specific program code. Master's thesis, TU Berlin, 2015.

[Lam93] Joachim Lambek. Least fixpoints of endofunctors of cartesian closed categories. *Mathematical Structures in Computer Science*, 3(2):229–257, 1993.

[LB17] Fengyun Liu and Eugene Burmako. Two approaches to portable macros. Technical report, EPFL, 2017.

[LHL17] Johannes Luong, Dirk Habich, and Wolfgang Lehner. AL: unified analytics in domain specific terms. In Tiark Rompf and Alexander Alexandrov, editors,

Proceedings of The 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, September 1, 2017, pages 7:1–7:9, 2017.

[MA86] Ernest G. Manes and Michael A. Arbib. *Algebraic Approaches to Program Ssemantics*. Texts and Monographs in Computer Science. 1986.

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

[MBB06] Erik Meijer, Brian Beckman, and Gavin M. Bierman. LINQ: reconciling object, relations and XML in the .net framework. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, page 706, 2006.

[MN06] Guido Moerkotte and Thomas Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 930–941, 2006.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[Pie91] Benjamin C. Pierce. *Basic category theory for computer scientists*. Foundations of computing. 1991.

[PLO17] Dmitry Petrashko, Ondrej Lhoták, and Martin Odersky. Miniphases: compilation using modular and efficient tree transformations. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 201–216, 2017.

[PSK17] Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. Quoted staged rewriting: a practical approach to library-defined optimizations. In Matthew

Flatt and Sebastian Erdweg, editors, *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, Vancouver, BC, Canada, October 23-24, 2017*, pages 131–145, 2017.

[PVSK18] Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. Unifying analytic and statically-typed quasiquotes. *PACMPL*, 2(POPL):13:1–13:33, 2018.

[RAM+12] Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. Scala-virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation*, 25(1):165–207, 2012.

[RO10] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In Eelco Visser and Jaakko Järvi, editors, *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*, pages 127–136, 2010.

[RO12] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In Philip A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1.*, pages 23–34, 1979.

[SBL+14] Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embedded Comput. Syst.*, 13(4s):134:1–134:25, 2014.

[Ski93a] D. B. Skillicorn. Structuring data parallelism using categorical data types. In *Proc. Workshop Programming Models for Massively Parallel Computers*, pages 110–115, September 1993.

[Ski93b] David B Skillicorn. The bird-meertens formalism as a parallel model. In *Software for Parallel Computation*, pages 120–133. 1993.

[SW95] Dan Suciu and Limsoon Wong. On two forms of structural recursion. In Georg Gottlob and Moshe Y. Vardi, editors, *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings*, volume 893 of *Lecture Notes in Computer Science*, pages 111–124, 1995.

[TBN91] Val Tannen, Peter Buneman, and Shamim A. Naqvi. Structural recursion as a query language. In Paris C. Kanellakis and Joachim W. Schmidt, editors, *Database Programming Languages: Bulk Types and Persistent Data. 3rd International Workshop, August 27-30, 1991, Nafplion, Greece, Proceedings*, pages 9–19, 1991.

[Tri91] Philip W. Trinder. Comprehensions, a query notation for dbpls. In Paris C. Kanellakis and Joachim W. Schmidt, editors, *Database Programming Languages: Bulk Types and Persistent Data. 3rd International Workshop, August 27-30, 1991, Nafplion, Greece, Proceedings*, pages 55–68, 1991.

[TS91] Val Tannen and Ramesh Subrahmanyam. Logical and computational aspects of programming with sets/bags/lists. In Javier Leach Albert, Burkhard Monien, and Mario Rodríguez-Artalejo, editors, *Automata, Languages and Programming, 18th International Colloquium, ICALP91, Madrid, Spain, July 8-12, 1991, Proceedings*, volume 510 of *Lecture Notes in Computer Science*, pages 60–75, 1991.

[TS00] Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[TSJ+09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[TW89] Phil Trinder and Philip Wadler. Improving list comprehension database queries. In *TENCON'89. Fourth IEEE Region 10 International Conference*, pages 186–192. IEEE, 1989.

[UG15] Alexander Ulrich and Torsten Grust. The flatter, the better: Query compilation based on the flattening transformation. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1421–1426, 2015.

[vdBKV03] Mark van den Brand, Paul Klint, and Jurgen J. Vinju. Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, 2003.

[Wad89] Philip Wadler. Theorems for free! In Joseph E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359, 1989.

[Wad90] Philip Wadler. Notes on monads and ringads. Internal document, Computing Science Dept. Glasgow University, September 1990.

[Wad92] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 1992.

[Wad95] Philip Wadler. How to declare an imperative. In John W. Lloyd, editor, *Logic Programming, Proceedings of the 1995 International Symposium, Portland, Oregon, USA, December 4-7, 1995*, pages 18–32, 1995.

[ZBW+12] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Åke Larson, Ronnie Chaiken, and Darren Shakib. SCOPE: parallel databases meet mapreduce. *VLDB J.*, 21(5):611–636, 2012.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

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.
CSE	**C**ommon **S**ubexpression **E**limination.
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.
sDSL	**S**tand-alone **D**omain **S**pecific **L**anguage.
SQL	**S**tructured **Q**uery **L**anguage.
SSA	**S**tatic **S**ingle **A**ssignment.
TQL	**T**raversal **Q**uery **L**anguage.
UDA	**U**ser-Defined **A**ggregate.
UDF	**U**ser-Defined **F**unction.
UDT	**U**ser-Defined **T**ype.