Reify Your Collection Queries for Modularity and Speed!

Paolo G. Giarrusso
Philipps University Marburg

Klaus Ostermann
Philipps University Marburg

Michael Eichberg
Software Technology Group, Technische Universität Darmstadt

Ralf Mitschke
Software Technology Group, Technische Universität Darmstadt

Tillmann Rendel
Philipps University Marburg

Christian Kästner
Carnegie Mellon University

ABSTRACT

Modularity and efficiency are often contradicting requirements, such that programers have to trade one for the other. We analyze this dilemma in the context of programs operating on collections. Performance-critical code using collections need often to be hand-optimized, leading to non-modular, brittle, and redundant code. In principle, this dilemma could be avoided by automatic collection-specific optimizations, such as fusion of collection traversals, usage of indexing, or reordering of filters. Unfortunately, it is not obvious how to encode such optimizations in terms of ordinary collection APIs, because the program operating on the collections is not reified and hence cannot be analyzed.

We propose SQuOpt, the Scala Query Optimizer—a *deep embedding* of the Scala collections API that allows such analyses and optimizations to be defined and executed within Scala, without relying on external tools or compiler extensions. SQuOpt provides the same "look and feel" (syntax and static typing guarantees) as the standard collections API. We evaluate SQuOpt by re-implementing several code analyses of the FindBugs tool using SQuOpt, show average speedups of 12x with a maximum of 12800x and hence demonstrate that SQuOpt can reconcile modularity and efficiency in real-world applications.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Query languages*; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.1.5 [**Programming Techniques**]: Object-oriented Programming

Keywords

Deep embedding; query languages; optimization; modularity

AOSD'13, March 24–29, 2013, Fukuoka, Japan.

1. INTRODUCTION

In-memory collections of data often need efficient processing. For on-disk data, efficient processing is already provided by database management systems (DBMS) thanks to their query optimizers, which support many optimizations specific to the domain of collections. Moving in-memory data to DBMSs, however, typically does not improve performance [30], and query optimizers cannot be reused separately since DBMS are typically monolithic and their optimizers deeply integrated. A few collection-specific optimizations, such as shortcut fusion [11], are supported by compilers for purely functional languages such as Haskell. However, the implementation techniques for those optimizations do not generalize to many other ones, such as support for indexes. In general, collection-specific optimizations are not supported by the general-purpose optimizers used by typical (JIT) compilers.

Therefore programmers, when needing collection-related optimizations, perform them manually. To allow that, they are often forced to perform manual inlining [24]. But manual inlining modifies source code by combining distinct functions together, while often distinct functions should remain distinct, because they deal with different concerns, or because one function need to be reused in a different context. In either case, manual inlining reduces modularity — defined here as the ability to abstract behavior in a separate function (possibly part of a different module) to enable reuse and improve understandability.

For these reasons, currently developers need to choose between modularity and performance, as also highlighted by Kiczales et al. [18] on a similar example. Instead, we envision that they should rely on an automatic optimizer performing inlining and collection-specific optimizations. They would then achieve both performance and modularity.[1]

One way to implement such an optimizer would be to extend the compiler of the language with a collection-specific optimizer, or to add some kind of external preprocessor to the language. However, such solutions would be rather

[1]In the terminology of Kiczales et al. [18], our goal is to be able to decompose different *generalized procedures* of a program according to its primary decomposition, while separating the handling of some performance concerns. To this end, we are modularizing these performance concerns into a metaprogramming-based optimization module, which we believe could be called, in that terminology, *aspect*.

brittle (for instance, they lack composability with other language extensions) and they would preclude optimization opportunities that arise only at runtime.

For this reason, our approach is implemented as an embedded domain-specific language, that is, as a regular library. We call this library SQuOpt, the Scala QUery OPTimizer. SQuOpt consists of a domain-specific language (DSL) for queries on collections based on the Scala collections API. This DSL is implemented as an embedded DSL (EDSL) for Scala. An expression in this EDSL produces at run time an *expression tree* in the host language: a data structure which represents the query to execute, similar to an abstract syntax tree (AST) or a query plan. Thanks to the extensibility of Scala, expressions in this language look almost identical to expressions with the same meaning in Scala. When executing the query, SQuOpt optimizes and compiles these expression trees for more efficient execution. Doing optimization at run time, instead of compile-time, avoids the need for control-flow analyses to determine which code will be actually executed [3], as we will see later.

We have choosen Scala [23] to implement our library for two reasons: (i) Scala is a good meta-language for embedded DSLs, because it is syntactically flexible and has a powerful type system, and (ii) Scala has a sophisticated collections library with an attractive syntax (for-comprehensions) to specify queries.

To evaluate SQuOpt, we study queries of the FindBugs tool [17]. We rewrote a set of queries to use the Scala collections API and show that modularization incurs significant performance overhead. Subsequently, we consider versions of the same queries using SQuOpt. We demonstrate that the automatic optimization can reconcile modularity and performance in many cases. Adding advanced optimizations such as indexing can even improve the performance of the analyses beyond the original non-modular analyses.

Overall, our main contributions are the following:

- We illustrate the tradeoff between modularity and performance when manipulating collections, caused by the lack of domain-specific optimizations (Sec. 2). Conversely, we illustrate how domain-specific optimizations lead to more readable and more modular code (Sec. 3).
- We present the design and implementation of SQuOpt, an embedded DSL for queries on collections in Scala (Sec. 4).
- We evaluate SQuOpt to show that it supports writing queries that are at the same time modular and fast. We do so by re-implementing several code analyses of the FindBugs tool. The resulting code is more modular and/or more efficient, in some cases by orders of magnitude. In these case studies, we measured average speedups of 12x with a maximum of 12800x (Sec. 5).

2. MOTIVATION

In this section, we show how the absense of collection-specific optimizations forces programmers to trade modularity against performance, which motivates our design of SQuOpt to resolve this conflict.

As our running example through the paper, we consider representing and querying a simple in-memory bibliography. A book has, in our schema, a title, a publisher and a list of authors. Each author, in turn, has a first and last name. We represent authors and books as instances of the Scala classes `Author` and `Book` shown in Fig. 1. The class declarations list the type of each field: Titles, publishers, and first and last names are all stored in fields of type `String`. The list of authors is stored in a field of type `Seq[Author]`, that is, a sequence of authors – something that would be more complex to model in a relational database. The code fragment also defines a collection of books named `books`.

```
package schema
case class Author(firstName: String, lastName: String)
case class Book(title: String, publisher: String,
  authors: Seq[Author])

val books: Set[Book] = Set(
  new Book("Compilers: Principles, Techniques and Tools",
       "Pearson Education",
       Seq(new Author("Alfred V.", "Aho"),
           new Author("Monica S.", "Lam"),
           new Author("Ravi", "Sethi"),
           new Author("Jeffrey D.", "Ullman"))
  /* other books ... */)
```

Figure 1: Definition of the schema and of some content.

```
case class BookData(title: String, authorName: String,
  coauthors: Int)

val records =
  for {
    book ← books
    if book.publisher == "Pearson Education"
    author ← book.authors
  } yield new BookData(book.title,
                author.firstName + " " +
                author.lastName,
                book.authors.size - 1)

def titleFilter(records: Set[BookData],
    keyword: String) =
  for {
    record ← records
    if record.title.contains(keyword)
  } yield (record.title, record.authorName)

val res = titleFilter(records, "Principles")
```

Figure 2: Our example query on the schema in Fig. 1, and a function which postprocesses its result.

As a common idiom to query such collections, Scala provides *for-comprehensions*. For instance, the for-comprehension computing `records` in Fig. 2 finds all books published by Pearson Education and yields, for each of those books, and for each of its authors, a record containing the book title, the full name of that author and the number of additional coauthors. The *generator* `book ← books` functions like a loop header: The remainder of the for-comprehension is executed once per book in the collection. Consequently, the *generator* `author ← book.authors` starts a nested loop. The return value of the for-comprehension is a collection of all yielded records. Note that if a book has multiple authors, this for-comprehensions will return multiple records relative to this book, one for each author.

We can further process this collection with another for-comprehension, possibly in a different module. For example, still in Fig. 2, the function `titleFilter` filters book titles

containing the word "Principles", and drops from each record the number of additional coauthors.

In Scala, the implementation of for-comprehensions is not fixed. Instead, the compiler desugars a for-comprehension to a series of API calls, and different collection classes can implement this API differently. Later, we will use this flexibility to provide an optimizing implementation of for-comprehensions, but in this section, we focus on the behavior of the standard Scala collections, which implement for-comprehensions as loops that create intermediate collections.

2.1 Optimizing by Hand

In the naive implementation in Fig. 2 different concerns are separated, hence it is modular. However, it is also inefficient. To execute this code, we first build the original collection and only later we perform further processing to build the new result; creating the intermediate collection at the interface between these functions is costly. Moreover, the same book can appear in `records` more than once if the book has more than one author, but all of these duplicates have the same title. Nevertheless, we test each duplicate title separately whether it contains the searched `keyword`. If books have 4 authors on average, this means a slowdown of a factor of 4 for the filtering step.

In general, one can only resolve these inefficiencies by manually optimizing the query; however, we will observe that these manual optimizations produce less modular code.[2]

To address the first problem above, that is, to avoid creating intermediate collections, we can manually inline `titleFilter` and `records`; we obtain two nested for-comprehensions. Furthermore, we can *unnest* the inner one [6].

To address the second problem above, that is, to avoid testing the same title multiple times, we *hoist* the filtering step, that is, we change the order of the processing steps in the query to first look for `keyword` within `book.title` and then iterate over the set of authors. This does not change the overall semantics of the query because the filter only accesses the title but does not depend on the author. In the end, we obtain the code in Fig. 3. The resulting query processes the title of each book only once. Since filtering in Scala is done lazily, the resulting query avoids building an intermediate collection.

This second optimization is only possible after inlining and thereby reducing the modularity of the code, because it mixes together processing steps from `titleFilter` and from the definition of `records`. Therefore, reusing the code creating records would now be harder.

To make `titleFilterHandOpt` more reusable, we could turn the publisher name into a parameter. However, the new versions of `titleFilter` cannot be reused as-is if some details of the inlined code change; for instance, we might need to filter publishers differently or not at all. On the other hand, if we express queries modularly, we might lose some opportunities for optimization. The design of the collections API, both in Scala and in typical languages, forces us to manually optimize our code by repeated inlining and subsequent application of query optimization rules, which leads to a loss of modularity.

[2]The existing Scala collections API supports optimization, for instance through non-strict variants of the query operators (called 'views' in Scala), but they can only be used for a limited set of optimizations, as we discuss in Sec. 6.

```
def titleFilterHandOpt(books: Set[Book],
                       publisher: String,
                       keyword: String) =
  for {
    book ← books
    if book.publisher == publisher &&
  book.title.contains(keyword)
    author ← book.authors
  } yield (book.title, author.firstName + " " +
           author.lastName)
val res = titleFilterHandOpt(books,
  "Pearson Education", "Principles")
```

Figure 3: Composition of queries in Fig. 2, after inlining, query unnesting and hoisting.

```
import squopt._
import schema.squopt._

val recordsQuery =
  for {
    book ← books.asSquopt
    if book.publisher ==# "Pearson Education"
    author ← book.authors
  } yield new BookData(book.title,
    author.firstName + " " + author.lastName,
    book.authors.size - 1)

// ...
val records = recordsQuery.eval

def titleFilterQuery(records: Exp[Set[BookData]],
                keyword: Exp[String]) = for {
  record ← records
  if record.title.contains(keyword)
} yield (record.title, record.authorName)
val resQuery = titleFilterQuery(recordsQuery, "Principles")
val res = resQuery.optimize.eval
```

Figure 4: SQuOpt version of Fig. 2; `recordQuery` contains a reification of the query, `records` its result.

3. AUTOMATIC OPTIMIZATION WITH SQUOPT

The goal of SQuOpt is to let programmers write queries modularly and at a high level of abstraction and deal with optimization by a dedicated domain-specific optimizer. In our concrete example, programmers should be able to write queries similar to the one in Fig. 2, but get the efficiency of the one in Fig. 3. To allow this, SQuOpt overloads for-comprehensions and other constructs, such as string concatenation with `+` and field access `book.author`. Our overloads of these constructs reify the query as an expression tree. SQuOpt can then optimize this expression tree and execute the resulting optimized query. Programmers explicitly trigger processing by SQuOpt, by adapting their queries as we describe in next subsection.

3.1 Adapting a Query

To use SQuOpt instead of native Scala queries, we first assume that the query does not use side effects and is thus *purely functional*. We argue that purely functional queries are more declarative. Side effects are used to improve performance, but SQuOpt makes that unnecessary through automatic optimizations. In fact, the lack of side effects enables more optimizations.

In Fig. 4 we show a version of our running example adapted to use SQuOpt. We first discuss changes to `records`. To

enable SQuOpt, a programmer needs to (a) import the SQuOpt library, (b) import some wrapper code specific to the types the collection operates on, in this case `Book` and `Author` (more about that later), (c) convert explicitly the native Scala collections involved to collections of our framework by a call to `asSquopt`, (d) rename a few operators such as `==` to `==#` (this is necessary due to some Scala limitations), and (e) add a separate step where the query is evaluated (possibly after optimization). All these changes are lightweight and mostly of a syntactic nature.

For parameterized queries like `titleFilter`, we need to also adapt type annotations. The ones in `titleFilterQuery` reveal some details of our implementation: Expressions that are reified have type `Exp[T]` instead of `T`. As the code shows, `resQuery` is optimized before compilation. This call will perform the optimizations that we previously did by hand and will return a query equivalent to that in Fig. 3, after verifying their safety conditions. For instance, after inlining, the filter `if book.title.contains(keyword)` does not reference `author`; hence, it is safe to hoist. Note that checking this safety condition would not be possible without reifying the predicate. For instance, it would not be sufficient to only reify the calls to the collection API, because the predicate is represented as a boolean function parameter. In general, our automatic optimizer inspects the whole reification of the query implementation to check that optimizations do not introduce changes in the overall result of the query and are therefore safe.

3.2 Indexing

SQuOpt also supports the transparent usage of indexes. Indexes can further improve the efficiency of queries, sometimes by orders of magnitude. In our running example, the query scans all books to look for the ones having the right publisher. To speed up this query, we can preprocess `books` to build an index, that is, a dictionary mapping, from each publisher to a collection of all the books it published. This index can then be used to answer the original query without scanning all books.

We construct a *query* representing the desired dictionary, and inform the optimizer that it should use this index where appropriate:

```
val idxByPublisher =
  books.asSquopt.indexBy(_.publisher)
Optimization.addIndex(idxByPublisher)
```

The `indexBy` collection method accepts a function that maps a collection element to a key; `coll.indexBy(key)` returns a dictionary mapping each key to the collection of all elements of `coll` having that key. Missing keys are mapped to an empty collection.[3] `Optimization.addIndex` simply preevaluates the index and updates a dictionary mapping the index to its preevaluated result.

A call to `optimize` on a query will then take this index into account and rewrite the query to perform index lookup instead of scanning, if possible. For instance, the code in Fig. 4 would be transparently rewritten by the optimizer to a query similar to the following:

```
val indexedQuery =
  for {
    book ← idxByPublisher("Pearson Education")
    author ← book.authors
  } yield new BookData(book.title, author.firstName
    + " " + author.lastName, book.authors.size - 1)
```

[3]For readers familiar with the Scala collection API, we remark that the only difference with the standard `groupBy` method is the handling of missing keys.

Since dictionaries in Scala are functions, in the above code, dictionary lookup on `idxByPublisher` is represented simply as function application. The above code iterates over books having the desired publisher, instead of scanning the whole library, and performs the remaining computation from the original query. Although the index use in the listing above is notated as `idxByPublisher("Pearson Education")`, only the cached result of evaluating the index is used when the query is executed, not the reified index definition.

This optimization could also be performed manually, of course, but the queries are on a higher abstraction level and more maintainable if indexing is defined separately and applied automatically. Manual application of indexing is a crosscutting concern because adding or removing an index affects potentially many queries. SQuOpt does not free the developer from the task of assessing which index will 'pay off' (we have not considered automatic index creation yet), but at least it becomes simple to add or remove an index, since the application of the indexes is modularized in the optimizer.

4. IMPLEMENTATION

After describing how to use SQuOpt, we explain how SQuOpt represents queries internally and optimizes them. We give only a brief overview of our implementation technique; it is described in more detail in a technical report that accompanies this paper [10].

4.1 Expression Trees

In order to analyze and optimize collection queries at runtime, SQuOpt reifies their syntactic structure as *expression trees*. The expression tree reflects the syntax of the query after desugaring, that is, after for-comprehensions have been replaced by API calls. For instance, `recordsQuery` from Fig. 4 points to the following expression tree (with some boilerplate omitted for clarity):

```
new FlatMap(
  new Filter(
    new Const(books),
    v2 ⇒ new Eq(new Book_publisher(v2),
                new Const("Pearson Education"))),
    v3 ⇒ new MapNode(
          new Book_authors(v3),
          v4 ⇒ new BookData(
                 new Book_title(v3),
                 new StringConcat(
                   new StringConcat(
                     new Author_firstName(v4),
                     new Const(" ")),
                   new Author_lastName(v4)),
                 new Plus(new Size(new Book_authors(v3)),
                          new Negate(new Const(1))))))
```

The structure of the for-comprehension is encoded with the `FlatMap`, `Filter` and `MapNode` instances. These classes correspond to the API methods that for-comprehensions get desugared to. SQuOpt arranges for the implementation of `flatMap` to construct a `FlatMap` instance, etc. The instances of the other classes encode the rest of the structure of the collection query, that is, which methods are called on which arguments. On the one hand, SQuOpt defines classes such as `Const` or `Eq` that are generic and applicable to all queries. On the other hand, classes such as `Book_publisher` cannot be

predefined, because they are specific to the user-defined types used in a query. SQuOpt provides a small code generator, which creates a case class for each method and field of a user-defined type. Functions in the query are represented by functions that create expression trees; representing functions in this way is frequently called higher-order abstract syntax [25].

We can see that the reification of this code corresponds closely to an abstract syntax tree for the code which is executed; however, many calls to specific methods, like `map`, are represented by special nodes, like `MapNode`, rather than as method calls. For the optimizer it becomes easier to match and transform those nodes than with a generic abstract syntax tree.

Nodes for collection operations are carefully defined by hand to provide them highly generic type signatures and make them reusable for all collection types. In Scala, collection operations are highly polymorphic; for instance, `map` has a single implementation working on all collection types, like `List`, `Set`, and we similarly want to represent all usages of `map` through instances of a single node type, namely `MapNode`. Having separate nodes `ListMapNode`, `SetMapNode` and so on would be inconvenient, for instance when writing the optimizer. However, `map` on a `List[Int]` will produce another `List`, while on a `Set` it will produce another `Set`, and so on for each specific collection type (in first approximation); moreover, this is guaranteed statically by the type of `map`. Yet, thanks to advanced typesystem features, `map` is defined only once avoiding redundancy, but has a type polymorphic enough to guarantee statically that the correct return value is produced. Since our tree representation is strongly typed, we need to have a similar level of polymorphism in `MapNode`. We achieved this by extending the techniques described by Odersky and Moors [22], as detailed in our technical report [10].

We get these expression trees by using Scala implicit conversions in a particular style, which we adopted from Rompf and Odersky [26]. Implicit conversions allow to add, for each method `A.foo(B)`, an overload of `Exp[A].foo(Exp[B])`. Where a value of type `Exp[T]` is expected, a value of type `T` can be used thanks to other implicit conversions, which wrap it in a `Const` node. The initial call of `asSquopt` triggers the application of the implicit conversions by converting the collection to the leaf of an expression tree.

It is also possible to call methods that do not return expression trees; however, such method calls would then only be represented by an opaque `MethodCall` node in the expression tree, which means that the code of the method cannot be considered in optimizations.

Crucially, these expression trees are generated at runtime. For instance, the first `Const` contains a reference to the actual collection of books to which `books` refers. If a query uses another query, such as `records` in Fig. 4, then the subquery is effectively *inlined*. The same holds for method calls inside queries: If these methods return an expression tree (such as the `titleFilterQuery` method in Fig. 4), then these expression trees are inlined into the composite query. Since the reification happens at runtime, it is not necessary to predict the targets of dynamically bound method calls: A new (and possibly different) expression tree is created each time a block of code containing queries is executed.

Hence, we can say that expression trees represent the computation which is going to be executed after inlining; control flow or virtual calls in the original code typically disappear—especially if they manipulate the query as a whole. This is typical of deeply embedded DSLs like ours, where code instead of performing computations produces a representation of the computation to perform [5, 3].

This inlining can duplicate computations; for instance, in this code:

```
val num: Exp[Int] = 10
val square = num * num
val sum = square + square
```

evaluating `sum` will evaluate `square` twice. Elliott et al. [5] and we avoid this using common-subexpression elimination.

4.2 Optimizations

Our optimizer currently supports several algebraic optimizations. Any query and in fact every reified expression can be optimized by calling the `optimize` function on it. The ability to optimize reified expressions that are not queries is useful; for instance, optimizing a function that produces a query is similar to a "prepared statement" in relational databases.

The optimizations we implemented are mostly standard in compilers [21] or databases:

- *Query unnesting* merges a nested query into the containing one [6, 14], replacing for instance

  ```
  for {val1 ← (for {val2 ← coll} yield f(val2))}
    yield g(val1)
  ```

  with

  ```
  for {val2 ← coll; val1 = f(val2)} yield g(val1)
  ```

- *Bulk operation fusion* fuses higher-order operators on collections.

- *Filter hoisting* tries to apply filters as early as possible; in database query optimization, it is known as selection pushdown. For filter hoisting, it is important that the full query is reified, because otherwise the dependencies of the filter condition cannot be determined.

- We reduce during optimization tuple/case class accesses: For instance, `(a, b)._1` is simplified to `a`. This is important because the produced expression does not depend on `b`; removing this false dependency can allow, for instance, a filter containing this expression to be hoisted to a context where `b` is not bound.

- *Indexing* tries to apply one or more of the available indexes to speed up the query.

- *Common subexpression elimination (CSE)* avoids that the same computation is performed multiple times; we use techniques similar to Rompf and Odersky [26].

- Smaller optimizations include constant folding, reassociation of associative operators and removal of identity maps (`coll.map(x ⇒ x)`, typically generated by the translation of for-comprehensions).

Each optimization is applied recursively bottom-up until it does not trigger anymore; different optimizations are composed in a fixed pipeline.

Optimizations are only guaranteed to be semantics-preserving if queries obey the restrictions we mentioned: for

instance, queries should not involve side-effects such as assignments or I/O, and all collections used in queries should implement the specifications stated in the collections API. Obviously the choice of optimizations involves many tradeoffs; for that reason we believe that it is all the more important that the optimizer is not hard-wired into the compiler but implemented as a library, with potentially many different implementations.

To make changes to the optimizer more practical, we designed our query representation so that optimizations are easy to express; restricting to pure queries also helps. For instance, filter fusion can be implemented simply as:[4]

```
val mergeFilters = ExpTransformer {
  case Sym(Filter(Sym(Filter(collection, pred2)), pred1)) ⇒
    coll.filter(x ⇒ pred2(x) && pred1(x))
}
```

The above code matches on reified expression of form `coll.filter(pred2).filter(pred1)` and rewrites it. A more complex optimization such as filter hoisting requires only 20 lines of code.

We have implemented a prototype of the optimizer with the mentioned optimizations. Many additional algebraic optimizations can be added in future work by us or others; a candidate would be loop hoisting, which moves out of loops arbitrary computations not depending on the loop variable (and not just filters). With some changes to the optimizer's architecture, it would also be possible to perform cost-based and dynamic optimizations.

4.3 Query Execution

Calling the `eval` method on a query will convert it to executable bytecode; this bytecode will be loaded and invoked by using Java reflection. We produce a thunk that, when evaluated, will execute the generated code.

In our prototype we produce bytecode by converting expression trees to Scala code and invoking on the result the Scala compiler, `scalac`. Invoking `scalac` is typically quite slow, and we currently use caching to limit this concern; however, we believe it is merely an engineering problem to produce bytecode directly from expression trees, just as compilers do.

Our expression trees contain native Scala values wrapped in `Const` nodes, and in many cases one cannot produce Scala program text evaluating to the same value. To allow executing such expression trees we need to implement cross-stage persistence (CSP): the generated code will be a function, accepting the actual values as arguments [26]. This allows sharing the compiled code for expressions which differ only in the embedded values.

More in detail, our compilation algorithm is as follows. (a) We implement CSP by replacing embedded Scala values by references to the function arguments; so for instance `List(1, 2, 3).map(x ⇒ x + 1)` becomes the function `(s1: List[Int], s2: Int) ⇒ s1.map(x ⇒ x + s2)`. (b) We look up the produced expression tree, together with the types of the constants we just removed, in a cache mapping to the generated classes. If the lookup fails we update the cache with the result of the next steps. (c) We apply CSE on the expression. (d) We convert the tree to code, compile it and load the generated code.

Preventing errors in generated code Compiler errors in generated code are typically a concern; with SQuOpt, however, they can only arise due to implementation bugs in SQuOpt (for instance in pretty-printing, which cannot be checked statically), so they do not concern users. Since our query language and tree representation are statically typed, type-incorrect queries will be rejected statically. For instance, consider again `idxByPublisher`, described previously:

```
val idxByPublisher =
  books.asSquopt.indexBy(_.publisher)
```

Since `Book.publisher` returns a `String`, `idxByPublisher` has type `Exp[Map[String, Book]]`. Looking up a key of the wrong type, for instance by writing `idxByPublisher(book)` where `book: Book`, will make `scalac` emit a static type error.

[4] `Sym` nodes are part of the boilerplate we omitted earlier.

5. EVALUATION

The key goals of SQuOpt are to reconcile *modularity* and *efficiency*. To evaluate this claim, we perform a rigorous performance evaluation of queries with and without SQuOpt. We also analyze modularization potential of these queries and evaluate how modularization affects performance (with and without SQuOpt).

We show that modularization introduces a significant slowdown. The overhead of using SQuOpt is usually moderate, and optimizations can compensate this overhead, remove the modularization slowdown and improve performance of some queries by orders of magnitude, especially when indexes are used.

5.1 Study Setup

Throughout the paper, we have already shown several compact queries for which our optimizations increase performance significantly compared to a naive execution. Since some optimizations change the complexity class of the query (e.g. by using an index), so the speedups grow with the size of the data. However, to get a more realistic evaluation of SQuOpt, we decided to perform an experiment with existing real-world queries.

As we are interested in both performance and modularization, we have a specification and three different implementations of each query that we need to compare:

(0) **Query specification:** We selected a set of existing real-world queries specified and implemented independently from our work and prior to it. We used only the specification of these queries.

(1) **Modularized Scala implementation:** We reimplemented each query as an expression on Scala collections — our baseline implementation. For modularity, we separated reusable domain abstractions into subqueries. We confirmed the abstractions with a domain expert and will later illustrate them to emphasize their general nature.

(2) **Hand-optimized Scala implementation:** Next, we asked a domain expert to performed manual optimizations on the modularized queries. The expert should perform optimizations, such as inlining and filter hoisting, where he could find performance improvements.

(3) **SQuOpt implementation:** Finally, we rewrote the modularized Scala queries from (1) as SQuOpt queries. The rewrites are of purely syntactic nature to use our library (as described in Sec. 3.1) and preserve the modularity of the queries.

Since SQuOpt supports executing queries with and without optimizations and indexes, we measured actually three different execution modes of the SQuOpt implementation:

(3^-) **SQuOpt without optimizer:** First, we execute the SQuOpt queries without performing optimization first, which should show the SQuOpt overhead compared to the modular Scala implementation (1). However, common-subexpression elimination is still used here, since it is part of the compilation pipeline. This is appropriate to counter the effects of excessive inlining due to using a deep embedding, as explained in Sec. 4.1.

(3^o) **SQuOpt with optimizer:** Next, we execute SQuOpt queries after optimization.

(3^x) **SQuOpt with optimizer and indexes:** Finally, we execute the queries after providing a set of indexes that the optimizer can consider.

In all cases, we measure query execution time for the generated code, excluding compilation: we consider this appropriate because the results of compilations are cached aggressively and can be reused when the underlying data is changed, potentially even across executions (even though this is not yet implemented), as the data is not part of the compiled code.

We use additional indexes in (3^x), but not in the hand-optimized Scala implementation (2). We argue that indexes are less likely to be applied manually, because index application is a crosscutting concern and makes the whole query implementation more complicated and less abstract. Still, we offer measurement (3^o) to compare the speedup without additional indexes.

This gives us a total of five settings to measure and compare (1, 2, 3^-, 3^o, and 3^x). Between them, we want to observe the following interesting performance ratios (speedups or slowdowns, computed through the indicated divisions):

(M) Modularization overhead (the relative performance difference between the modularized and the hand-optimized Scala implementation: 1/2).

(S) SQuOpt overhead (the overhead of executing unoptimized SQuOpt queries: $1/3^-$; smaller is better).

(H) Hand-optimization challenge (the performance overhead of our optimizer against hand-optimizations of a domain expert: $2/3^o$; bigger is better). This overhead is partly due to the SQuOpt overhead (S) and partly to optimizations which have not been automated or have not been effective enough. This comparison excludes the effects of indexing, since this is an optimization we did not perform by hand; we also report **(H')** $= 2/3^x$, which includes indexing.

(O) Optimization potential (the speedup by optimizing modularized queries: $1/3^o$; bigger is better).

(X) Index influence (the speedup gained by using indexes: $3^o/3^x$) (bigger is better).

(T) Total optimization potential with indexes ($1/3^x$; bigger is better), which is equal to $(O) \times (X)$.

In Figure 5, we provide an overview of the setup. We made our raw data available and our results reproducible [31].[5]

[5]Data available at: `http://www.informatik.uni-marburg.de/~pgiarrusso/SQuOpt`

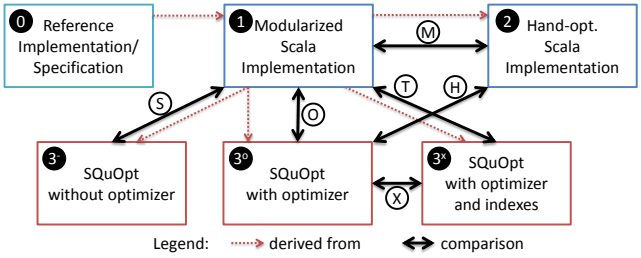

Figure 5: Measurement Setup: Overview

Abstraction	Used
All fields in all class files	4
All methods in all class files	3
All method bodies in all class files	3
All instructions in all method bodies and their bytecode index	5
Sliding window (size n) over all instructions (and their index)	3

Table 1: Description of abstractions removed during hand-optimization and number of queries where the abstraction is used (and optimized away).

5.2 Experimental Units

As experimental units, we sampled a set of queries on code structures from FindBugs 2.0 [17]. FindBugs is a popular bug-finding tool for Java Bytecode available as open source. To detect instances of bug patterns, it queries a structural in-memory representation of a code base (extracted from bytecode). Concretely, a single loop traverses each class and invokes all visitors (implemented as listeners) on each element of the class. Many visitors, in turn, perform activities concerning multiple bug detectors which are fused together. An extreme example is that, in FindBugs, query 4 is defined in class `DumbMethods` together with other 41 bug detectors for distinct types of bugs. Typically a bug detector is furthermore scattered across the different methods of the visitor, which handle different elements of the class. We believe this architecture has been chosen to achieve good performance; however, we do not consider such manual fusion of distinct bug detectors together as modular. We selected queries from FindBugs because they represent typical non-trivial queries on in-memory collections and because we believe our framework allows expressing them more modularly.

We sampled queries in two batches. First, we manually selected 8 queries (from approx. 400 queries in FindBugs), chosen mainly to evaluate the potential speedups of indexing (queries that primarily looked for declarations of classes, methods, or fields with specific properties, queries that inspect the type hierarchy, and queries that required analyzing methods implementation). Subsequently, we *randomly* selected a batch of 11 additional queries. The batch excluded queries that rely on control-/dataflow analyses (i.e., analyzing the effect of bytecode instructions on the stack), due to limitations of the bytecode tookit we use. In total, we have 19 queries as listed in Table 2 (the randomly selected queries are marked with the superscript R).

We implemented each query three times (see implementations (1)–(3) in Sec. 5.1) following the specifications given in

Id	Description	Performance (ms)					Performance ratios		
		1	2	3^-	3^o	3^x	M (1/2)	H ($2/3^o$)	T ($1/3^x$)
1	Covariant compareTo() defined	1.1	1.3	0.85	0.26	0.26	0.9	5.0	4.4
2	Explicit garbage collection call	496	258	1176	1150	52	1.9	0.2	9.5
3	Protected field in final class	11	1.1	11	1.2	1.2	10.0	1.0	9.8
4	Explicit runFinalizersOnExit() call	509	262	1150	1123	10.0	1.9	0.2	51
5	clone() defined in non-Cloneable class	29	14	55	46	0.47	2.1	0.3	61
6	Covariant equals() defined	29	15	23	9.7	0.20	1.9	1.6	147
7	Public finalizer defined	29	12	28	8.0	0.03	2.3	1.5	1070
8	Dubious catching of IllegalMonitorStateException	82	72	110	28	0.01	1.1	2.6	12800
9^R	Uninit. field read during construction of super	896	367	3017	960	960	2.4	0.4	0.9
10^R	Mutable static field declared public	9527	9511	9115	9350	9350	1.0	1.0	1.0
11^R	Refactor anon. inner class to static	8804	8767	8718	8700	8700	1.0	1.0	1.0
12^R	Inefficient use of toArray(Object[])	3714	1905	4046	3414	3414	2.0	0.6	1.1
13^R	Primitive boxed and unboxed for coercion	3905	1672	5044	3224	3224	2.3	0.5	1.2
14^R	Double precision conversion from 32 bit	3887	1796	5289	3010	3010	2.2	0.6	1.3
15^R	Privileged method used outside doPrivileged	505	302	1319	337	337	1.7	0.9	1.5
16^R	Mutable public static field should be final	13	6.2	12	7.0	7.0	2.0	0.9	1.8
17^R	Serializable class is member of non-ser. class	12	0.77	0.94	1.8	1.8	16	0.4	6.9
18^R	Swing methods used outside Swing thread	577	53	1163	45	45	11	1.2	13
19^R	Finalizer only calls super class finalize	55	13	73	11	0.10	4.4	1.1	541

Table 2: Performance results. As in in Sec. 5.1, (1) denotes the modular Scala implementation, (2) the hand-optimized Scala one, and (3^-), (3^o), (3^x) refer to the SQuOpt implementation when run, respectively, without optimizations, with optimizations, with optimizations and indexing. Queries marked with the R superscript were selected by random sampling.

	M (1/2)	S ($1/3^-$)	H ($2/3^o$)	H' ($2/3^x$)	O ($1/3^o$)	X ($3^o/3^x$)	T ($1/3^x$)
Geometric means of performance ratios	2.4x	1.2x	0.8x	5.1x	1.9x	6.3x	12x

Table 3: Average performance ratios. This table summarizes all interesting performance ratios across all queries, using the geometric mean [7]. The meaning of speedups is discussed in Sec. 5.1.

```
for {
  classFile ← classFiles.asSquopt
  method ← classFile.methods
  if method.isAbstract && method.name ==# "equals" &&
  method.descriptor.returnType ==# BooleanType
  parameterTypes ← Let(method.descriptor.parameterTypes)
  if parameterTypes.length ==# 1 && parameterTypes(0) ==#
  classFile.thisClass
} yield (classFile, method)
```

Figure 6: Find covariant `equals` methods.

the FindBugs documentation (0). Instead of using a hierarchy of visitors as the original implementations of the queries in FindBugs, we wrote the queries as for-comprehensions in Scala on an in-memory representation created by the Scala toolkit BAT.[6] BAT in particular provides comprehensive support for writing queries against Java bytecode in an idiomatic way. We exemplify an analysis in Fig. 6: It detects all co-variant `equals` methods in a project by iterating over all class files (line 2) and all methods, searching for methods named "`equals`" that return a boolean value and define a single parameter of the type of the current class.

Abstractions In the reference implementations (1), we identified several reusable abstractions as shown in Table 1. The reference implementations of all queries except 17^R use exactly one of these abstractions, which encapsulate the main loops of the queries.

Indexes For executing (3^x) (SQuOpt with indexes), we have constructed three indexes to speed up navigation over the queried data of queries 1–8: Indexes for method name, exception handlers, and instruction types. We illustrate the implementation of the method-name index in Fig. 7: it produces a collection of all methods and then indexes them using `indexBy`; its argument extracts from an entry the key, that is the method name. We selected which indexes to implement using guidance from SQuOpt itself; during optimizations, SQuOpt reports which indexes it could have applied to the given query. Among those, we tried to select indexes giving a reasonable compromise between construction cost and optimization speedup. We first measured the construction cost of these indexes:

Index	Elapsed time (ms)
Method name	97.99±2.94
Exception handlers	179.29±3.21
Instruction type	4166.49±202.85

For our test data, index construction takes less than 200 ms for the first two indexes, which is moderate compared to the time for loading the bytecode in the BAT representa-

[6] `http://github.com/Delors/BAT`

```
val methodNameIdx: Exp[Map[String, Seq[(ClassFile, Method)]]] =
  (for {
  classFile ← classFiles.asSquopt
  method ← classFile.methods
} yield (classFile, method)).indexBy(entry ⇒ entry._2.name)
```

Figure 7: A simple index definition

tion (4755.32 ± 141.66). Building the instruction index took around 4 seconds, which we consider acceptable since this index maps each type of instruction (e.g. `INSTANCEOF`) to a collection of all bytecode instructions of that type.

5.3 Measurement Setup

To measure performance, we executed the queries on the preinstalled JDK class library (`rt.jar`), containing 58M of uncompressed Java bytecode. We also performed a preliminary evaluation by running queries on the much smaller ScalaTest library, getting comparable results that we hence do not discuss. Experiments were run on a 8-core Intel Core i7-2600, 3.40 GHz, with 8 GB of RAM, running Scientific Linux release 6.2. The benchmark code itself is single-threaded, so it uses only one core; however the JVM used also other cores to offload garbage collection. We used the preinstalled OpenJDK Java version 1.7.0_05-icedtea and Scala 2.10.0-M7.

We measure steady-state performance as recommended by Georges et al. [9]. We invoke the JVM $p = 15$ times; at the beginning of each JVM invocation, all the bytecode to analyze is loaded in memory and converted into BAT's representation. In each JVM invocation, we iterate each benchmark until the variations of results becomes low enough. We measure the variations of results through the coefficient of variation (CoV; standard deviation divided by the mean). Thus, we iterate each benchmark until the CoV in the last $k = 10$ iterations drops under the threshold $\theta = 0.1$, or until we complete $q = 50$ iterations. We report the arithmetic mean of these measurements (and also report the usually low standard deviation on our web page).

5.4 Results

Correctness We machine-checked that for each query, all variants in Table 2 agree.

Modularization Overhead We first observe that performance suffers significantly when using the abstractions we described in Table 1. These abstractions, while natural in the domain and in the setting of a declarative language, are not idiomatic in Java or Scala because, without optimization, they will obviously lead to bad performance. They are still useful abstractions from the point of view of modularity, though—as indicated by Table 1—and as such it would be desirable if one could use them without paying the performance penalty.

Scala Implementations vs. FindBugs Before actually comparing between the different Scala and SQuOpt implementations, we first ensured that the implementations are comparable to the original FindBugs implementation. A direct comparison between the FindBugs reference implementation and any of our implementations is not possible in a rigorous and fair manner. FindBugs bug detectors are not fully modularized, therefore we cannot reasonably isolate the implementation of the selected queries from support code. Furthermore, the architecture of the implementation has many differences that affect performance: among others, FindBugs also uses multithreading. Moreover, while in our case each query loops over all classes, in FindBugs, as discussed above, a single loop considers each class and invokes all visitors (implemented as listeners) on it.

We measured *startup performance* [9], that is the performance of running the queries only once, to minimize the effect of compiler optimizations. We setup our SQuOpt-based analyses to only perform optimization and run the optimized query. To setup FindBugs, we manually disabled all unrelated bug detectors; we also made the modified FindBugs source code available. The result is that the performance of the Scala implementations of the queries (3^-) has performance of the same order of magnitude as the original FindBugs queries – in our tests, the SQuOpt implementation was about twice as fast. However, since the comparison cannot be made fair, we refrained from a more detailed investigation.

SQuOpt Overhead and Optimization Potential We present the results of our benchmarks in Table 2. Column names refer to a few of the definitions described above; for readability, we do not present all the ratios previously introduced for each query, but report the raw data. In Table 3, we report the geometric mean [7] of each ratio, computed with the same weight for each query.

We see that, in its current implementation, SQuOpt can cause a overhead S ($1/3^-$) up to 3.4x. On average SQuOpt queries are 1.2x faster. These differences are due to minor implementation details of certain collection operators. For query 18^R, instead, we have that the the basic SQuOpt implementation is 12.9x faster and are investigating the reason; we suspect this might be related to the use of pattern matching in the original query.

As expected, not all queries benefit from optimizations; out of 19 queries, optimization affords for 15 of them significant speedups ranging from a 1.2x factor to a 12800x factor; 10 queries are faster by a factor of at least 5. Only queries 10^R, 11^R and 12^R fail to recover any modularization overhead.

We have analyzed the behavior of a few queries after optimization, to understand why their performance has (or has not) improved.

Optimization makes query 17^R slower; we believe this is because optimization replaces filtering by lazy filtering, which is usually faster, but not here. Among queries where indexing succeeds, query 2 has the least speedup. After optimization, this query uses the instruction-type index to find all occurrences of invocation opcodes (`INVOKESTATIC` and `INVOKEVIRTUAL`); after this step the query looks, among those invocations, for ones targeting `runFinalizersOnExit`. Since invocation opcodes are quite frequent, the used index is not very specific, hence it allows for little speedup (9.5x). However no other index applies to this query; moreover, our framework does not maintain any selectivity statistics on indexes to predict these effects. Query 19^R benefits from indexing without any specific tuning on our part, because it looks for implementations of `finalize` with some characteristic, hence the highly selective method-name index applies. After optimization, query 8 becomes simply an index lookup on the index for exception handlers, looking for handlers of `IllegalMonitorStateException`; it is thus not surprising that its speedup is thus extremely high (12800x). This speedup relies on an index which is specific for this kind of query, and building this index is slower than executing the unoptimized query. On the other hand, building this index is entirely appropriate in a situation where similar queries are

common enough. Similar considerations apply to usage of indexing in general, similarly to what happens in databases.

Optimization Overhead The current implementation of the optimizer is not yet optimized for speed (of the optimization algorithm). For instance, expression trees are traversed and rebuilt completely once for each transformation. However, the optimization overhead is usually not excessive and is 54.8 ± 85.5 ms, varying between 3.5 ms and 381.7 ms (mostly depending on the query size).

Limitations Although many speedups are encouraging, our optimizer is currently a proof-of-concept and we experienced some limitations:

- In a few cases hand-optimized queries are still faster than what the optimizer can produce. We believe these problems could be addressed by adding further optimizations.
- Our implementation of indexing is currently limited to immutable collections. For mutable collections, indexes must be maintained incrementally. Since indexes are defined as special queries in SQuOpt, incremental index maintenance becomes an instance of incremental maintenance of query results, that is, of incremental view maintenance. We plan to support incremental view maintenance as part of future work; however, indexing in the current form is already useful, as illustrated by our experimental results.

Threats to Validity With rigorous performance measurements and the chosen setup, our study was setup to maximize internal and construct validity. Although we did not involve an external domain expert and we did not compare the results of our queries with the ones from FindBugs (except while developing the queries), we believe that the queries adequately represent the modularity and performance characteristics of FindBugs and SQuOpt. However, since we selected only queries from a single project, external validity is limited. While we cannot generalize our results beyond FindBugs yet, we believe that the FindBugs queries are representative for complex in-memory queries performed by applications.

Summary We demonstrated on our real-world queries that relying on declarative abstractions in collection queries often causes a significant slowdown. As we have seen, using SQuOpt without optimization, or when no optimizations are possible, usually provides performance comparable to using standard Scala; however, SQuOpt optimizations can in most cases remove the slowdown due to declarative abstractions. Furthermore, relying on indexing allows to achieve even greater speedups while still using a declarative programming style. Some implementation limitations restrict the effectiveness of our optimizer, but since this is a preliminary implementation, we believe our evaluation shows the great potential of optimizing queries to in-memory collections.

6. RELATED WORK

This paper builds on prior work on language-integrated queries, query optimization, techniques for DSL embedding, and other works on code querying.

Language-Integrated Queries Microsoft's Language-Integrated Query technology (Linq) [20, 2] is similar to our work in that it also reifies queries on collections to enable analysis and optimization. Such queries can be executed against a variety of backends (such as SQL databases or in-memory objects), and adding new back-ends is supported. Its implementation uses *expression trees*, a compiler-supported implicit conversion between expressions and their reification as a syntax tree. There are various major differences, though. First, the support for expression trees is hard-coded into the compiler. This means that the techniques are not applicable in languages that do not explicitly support expression trees. More importantly, the way expression trees are created in Linq is generic and fixed. For instance, it is not possible to create different tree nodes for method calls that are relevant to an analysis (such as the `map` method) than for method calls that are irrelevant for the analysis (such as the `toString` method). For this reason, expression trees in Linq cannot be customized to the task at hand and contain too much low-level information. It is well-known that this makes it quite hard to implement programs operating on expression trees [4].

Linq queries can also not easily be decomposed and modularized. For instance, consider the task of refactoring the filter in the query `from x in y where x.z == 1 select x` into a function. Defining this function as `bool comp(int v) { return v == 1; }` would destroy the possibility of analyzing the filter for optimization, since the resulting expression tree would only contain a reference to an opaque function. The function could be declared as returning an expression tree instead, but then this function could not be used in the original query anymore, since the compiler expects an expression of type `bool` and not an expression tree of type `bool`. It could only be integrated if the expression tree of the original query is created by hand, without using the built-in support for expression trees.

Although queries against in-memory collections could theoretically also be optimized in Linq, the standard implementation, Linq2Objects, performs no optimizations.

A few optimized embedded DSLs allow executing queries or computations on distributed clusters. DryadLINQ [35], based on Linq, optimizes queries for distributed execution. It inherits Linq's limitations and thus does not support decomposing queries in different modules. Modularizing queries is supported instead by FlumeJava [3], another library (in Java) for distributed query execution. However, FlumeJava cannot express many optimizations because its representation of expressions is more limited; also, its query language is more cumbersome. Both problems are rooted in Java's limited support for embedded DSLs. Other embedded DSLs support parallel platforms such as GPUs or many-core CPUs, such as Delite [28].

Willis et al. [33, 34] add first-class queries to Java through a source-to-source translator and implement a few selected optimizations, including join order optimization and incremental maintenance of query results. They investigate how well their techniques apply to Java programs, and they suggest that programmers use manual optimizations to avoid expensive constructs like nested loops. While the goal of these works is similar to ours, their implementation as an external source-to-source-translator makes the adoption, extensibility, and composability of their technique difficult.

There have been many approaches for a closer integration of SQL queries into programs, such as HaskellDB [19] (which also inspired Linq), or Ferry [15] (which moves part of a program execution to a database). In Scala, there are also

APIs which integrate SQL queries more closely such as Slick.[7] Its frontend allows to define and combine type-safe queries, similarly to ours (also in the way it is implemented). However, the language for defining queries maps to SQL, so it does not support nesting collections in other collections (a feature which simplified our example in Sec. 2), nor distinguishes statically between different kinds of collections, such as `Set` or `Seq`. Based on Ferry, ScalaQL [8] extends Scala with a compiler-plugin to integrate a query language on top of a relational database. The work by Spiewak and Zhao [29] is unrelated to [8] but also called ScalaQL. It is similar to our approach in that it also proposes to reify queries based on for-comprehensions, but it is not clear from the paper how the reification works.[8]

Query Optimization Query optimization on relational data is a long-standing issue in the database community, but there are also many works on query optimization on objects [6, 13]. Compared to these works, we have only implemented a few simple query optimizations, so there is potential for further improvement of our work by incorporating more advanced optimizations.

Scala and DSL Embedding Technically, our implementation of SQuOpt is a deep embedding of a part of the Scala collections API [22]. Deep embeddings were pionereed by Leijen and Meijer [19] and Elliott et al. [5]. The technical details of the embedding are not the main topic of this paper; we are using some of the Scala techniques presented by Rompf and Odersky [26] for using implicits and for adding infix operators to a type. Similar to Rompf and Odersky [26], we also use the Scala compiler on-the-fly. A plausible alternative backend for SQuOpt would have been to use Delite [27], a framework for building highly efficient DSLs in Scala. Using this framework, in concurrent work, Rompf et al. [28] also optimize collection queries; while their work allows for imperative programs, they do not support embedding arbitrary libraries in an automated way. On the other hand, they can reuse support for automatic parallelization and multiple platforms present in Delite. Ackermann et al. [1] present Jet, which also optimizes collection queries but targets MapReduce-style computations in a distributed environment. Moreover, both works do not apply typical database optimizations such as indexing or filter hoisting.

We regard the Scala collections API [22] as a shallowly embedded query DSL. Query operators immediately perform collection operations when called, so that it is not possible to optimize queries before execution. In addition to these eager query operators, the Scala collections API also provides *views* to create lazy collections. Views are somewhat similar to SQuOpt in that they reify query operators as data structures and interpret them later. However, views are not used for automatic query optimization, but for explicitly changing the evaluation order of collection processing. Unfortunately, views are not suited as a basis for the implementation of SQuOpt because they only reify the outermost pipeline of collection operators, whereas nested collection operators as well as other Scala code in queries, such as filter predicates or `map` and `flatMap` arguments, are only shallowly embedded. Deep embedding of the whole query is necessary for many optimizations, as discussed in Sec. 3.

Code Querying In our evaluation we explore the usage of SQuOpt to express queries on code and re-implement a subset of the FindBugs [17] analyses. There are various other specialized code query languages such as CodeQuest [16] or D-CUBED [32]. Since these are special-purpose query languages that are not embedded into a host language, they are not directly comparable to our approach.

[7] `http://slick.typesafe.com/`

[8] We contacted the authors; they were not willing to provide more details or the sources of their approach.

7. FUTURE WORK

As part of future work we plan to add support for *incremental view maintenance* [12] to SQuOpt. This would allow, for instance, to update incrementally both indexes and query results.

To make our DSL more convenient to use, it would be useful to use the virtualized pattern matcher of Scala 2.10, when it will be more robust, to add support for pattern matching in our virtualized queries.

Finally, while our optimizations are type-safe, as they rewrite an expression tree to another of the same type, currently the Scala type-checker cannot verify this statically, because of its limited support for GADTs. Solving this problem conveniently would allow checking statically that transformations are safe and make developing them easier.

8. CONCLUSIONS

We have illustrated the tradeoff between performance and modularity for queries on in-memory collections. We have shown that it is possible to design a deep embedding of a version of the collections API which reifies queries and can optimize them at runtime. Writing queries using this framework is, except minor syntactic details, the same as writing queries using the collection library, hence the adoption barrier to using our optimizer is low.

Our evaluation shows that using abstractions in queries introduces a significant performance overhead with native Scala code, while SQuOpt, in most cases, makes the overhead much more tolerable or removes it completely. Optimizations are not sufficient on some queries, but since our optimizer is a proof-of-concept with many opportunities for improvement, we believe a more elaborate version will achieve even better performance and reduce these limitations.

Acknowledgements The authors thank Sebastian Erdweg for helpful discussions on this project, Katharina Haselhorst for help implementing the code generator, and the anonymous reviewers, Jacques Carette and Karl Klose for their helpful comments on this paper. This work is supported in part by the European Research Council, grant #203099 "ScalPL".

References

[1] S. Ackermann, V. Jovanovic, T. Rompf, and M. Odersky. Jet: An embedded DSL for high performance big data processing. In *Int'l Workshop on End-to-end Management of Big Data (BigData)*, 2012.

[2] G. M. Bierman, E. Meijer, and M. Torgersen. Lost in translation: formalizing proposed extensions to C#. In *OOPSLA*, pages 479–498. ACM, 2007.

[3] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375. ACM, 2010.

[4] O. Eini. The pain of implementing LINQ providers. *Commun. ACM*, 54(8):55–61, 2011.

[5] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *JFP*, 13(2):455–481, 2003.

[6] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Systems (TODS)*, 25:457–516, 2000.

[7] P. J. Fleming and J. J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, Mar. 1986.

[8] M. Garcia, A. Izmaylova, and S. Schupp. Extending Scala with database query capability. *Journal of Object Technology*, 9(4):45–68, 2010.

[9] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA*, pages 57–76. ACM, 2007.

[10] P. G. Giarrusso, K. Ostermann, M. Eichberg, R. Mitschke, T. Rendel, and C. Kästner. Reify your collection queries for modularity and speed! *CoRR*, abs/1210.6284, 2012. URL `http://arxiv.org/abs/1210.6284`.

[11] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *FPCA*, pages 223–232. ACM, 1993.

[12] D. Gluche, T. Grust, C. Mainberger, and M. Scholl. Incremental updates for materialized OQL views. In *Deductive and Object-Oriented Databases*, volume 1341 of *LNCS*, pages 52–66. Springer, 1997.

[13] T. Grust. *Comprehending queries*. PhD thesis, University of Konstanz, 1999.

[14] T. Grust and M. H. Scholl. How to comprehend queries functionally. *Journal of Intelligent Information Systems*, 12:191–218, 1999.

[15] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. FERRY: database-supported program execution. In *Proc. Int'l SIGMOD Conf. on Management of Data (SIGMOD)*, pages 1063–1066. ACM, 2009.

[16] E. Hajiyev, M. Verbaere, and O. de Moor. *CodeQuest*: Scalable source code queries with Datalog. In *ECOOP*, pages 2–27. Springer, 2006.

[17] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[18] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[19] D. Leijen and E. Meijer. Domain specific embedded compilers. In *DSL*, pages 109–122. ACM, 1999.

[20] E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling objects, relations and XML in the .NET framework. In *Proc. Int'l SIGMOD Conf. on Management of Data (SIGMOD)*, page 706. ACM, 2006.

[21] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. ISBN 1-55860-320-4.

[22] M. Odersky and A. Moors. Fighting bit rot with types (experience report: Scala collections). In *IARCS Conf. Foundations of Software Technology and Theoretical Computer Science*, volume 4, pages 427–451, 2009.

[23] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Inc, 2 edition, 2011.

[24] S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *JFP*, 12(4-5):393–434, 2002.

[25] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI*, pages 199–208. ACM, 1988.

[26] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, pages 127–136. ACM, 2010.

[27] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented DSLs. In *DSL*, pages 93–117, 2011.

[28] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *POPL*, pages 497–510. ACM, 2013.

[29] D. Spiewak and T. Zhao. ScalaQL: Language-integrated database queries for Scala. In *Proc. Conf. Software Language Engineering (SLE)*, 2009.

[30] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, pages 1150–1160. VLDB Endowment, 2007.

[31] J. Vitek and T. Kalibera. Repeatability, reproducibility, and rigor in systems research. In *Proc. Int'l Conf. Embedded Software (EMSOFT)*, pages 33–38. ACM, 2011.

[32] P. Węgrzynowicz and K. Stencel. The good, the bad, and the ugly: three ways to use a semantic code query system. In *OOPSLA*, pages 821–822. ACM, 2009.

[33] D. Willis, D. Pearce, and J. Noble. Efficient object querying for Java. In *ECOOP*, pages 28–49. Springer, 2006.

[34] D. Willis, D. J. Pearce, and J. Noble. Caching and incrementalisation in the Java Query Language. In *OOPSLA*, pages 1–18. ACM, 2008.

[35] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. Conf. Operating systems design and implementation*, OSDI'08, pages 1–14. USENIX Association, 2008.