Automatic Improvement of Apache Spark Queries using Semantics-preserving Program Reduction

Zoltan A. Kocsis[1], John H. Drake[2], Douglas Carson[3] and Jerry Swan[4]

[1]Computing Science and Mathematics, University of Stirling, UK
zoltan.kocsis@cs.stir.ac.uk
[2]The OR Group, Queen Mary University of London, UK
j.drake@qmul.ac.uk
[3]Keysight Technologies, Edinburgh, UK
douglas_carson@keysight.com
[4]Department of Computer Science, University of York, UK
jerry.swan@york.ac.uk

Abstract. Apache Spark is a popular framework for large-scale data analytics. Unfortunately, Spark's performance can be difficult to optimise, since queries freely expressed in source code are not amenable to traditional optimisation techniques. This article describes Hylas, a tool for automatically optimising Spark queries embedded in source code via the application of semantics-preserving transformations. The transformation method is inspired by functional programming techniques of "deforestation', which eliminate intermediate data structures from a computation. This contrasts with approaches defined entirely within structured query formats such as Spark SQL. Hylas can identify certain computationally expensive operations and ensure that performing them creates no superfluous data structures. This optimisation leads to significant improvements in execution time, with over 10,000 times improvement observed in some cases.

Keywords: Apache Spark, Search-based Software Engineering, Program Transformation, Query Optimisation, Automatic Improvement Programming, Genetic Improvement

1 Introduction

There is a burgeoning trend for large scale data analysis, driven in part by the rise in the use of social media and the Internet of Things [1]. Deciding how best to exploit such voluminous data is driving widespread activity in Big Data analytics. From the practitioner's perspective, the process of Big Data analytics is still a costly endeavour in terms of time and effort [2]. As a result, it is crucial that operations on large data sets are performed efficiently.

Published in *GECCO 2016 Companion - Proceedings of the 2016 Genetic and Evolutionary Computation Conference*, New York: ACM. 2nd International Genetic Improvement Workshop 2016: An International Workshop on the Repair and Optimisation of Software using Computational Search, 20.7.2016 - 20.7.2016, Denver, CO, USA, pp. 1141-1146.
https://doi.org/10.1145/2908961.2931692

Apache Spark[1] is a popular open source cluster computing framework maintained by the Apache Software Foundation, and is an increasingly popular choice for Big Data analytics. Spark is written in the Scala programming language [3] and runs on the Java Virtual Machine [4]. The framework is organized around special in-memory data structures, known as Resilient Distributed Datasets (RDDs) [4]. RDDs avoid the costly read/write cycles incurred by other methods, where replication of data is required when sharing data between multiple computations. The ability to share data between executions is desirable in many iterative machine learning and data mining methods, where a substantial number of queries are run on the same set (or subset) of data. Despite the fact that data kept in-memory is somewhat more volatile than that stored on disk, RDDs have been designed with fault tolerance in mind and are recoverable in the event of failure. The Spark interface to RDDs consists of several well-known functional programming transformations (e.g. aggregate, filter, map and reduce) since these are more easily parallelized than their imperative equivalents. Generally, Spark queries are sequences of such transformations (which construct new RDDs), terminated by one or more actions (which return values).

Although much of Spark's popularity is due to the speed benefits gained via in-memory data processing, the Spark framework is not amenable to common query optimisation techniques such as those of relational algebra [5]. Since Spark queries are written in a general-purpose programming language (one of Scala, Java or Python), determining the underlying relational operations becomes extremely difficult [6]. To avoid this difficulty, previous work focused on special-purpose declarative data manipulation languages like Spark SQL, which allow for the use of these traditional techniques. Unfortunately, these optimisers cannot be used to improve existing programs which were not written with Spark SQL in mind [7].

Long before Spark existed, functional programmers faced a rather similar issue: programs written in functional style, using high-level functions to encapsulate common patterns of data-handling, frequently resulted in a large number of superfluous intermediate data structures [8]. Several so-called "deforestation" rules were proposed to automatically eliminate these superfluous structures.

In this article, we apply deforestation rules to reduce Spark queries to functionally equivalent forms that are more efficient, reducing the overall time taken to execute such queries on average for the cases tested. The key insight of the approach presented here is that RDDs are also intermediate data structures in a functional language. This makes it possible to apply similar deforestation rules to existing programs which manipulate RDDs in a less constrained fashion than approaches such as Spark SQL.

2 Related work

Within the field of Search Based Software Engineering (SBSE), there is widespread interest in the use of search techniques to improve functional and/or non-functional

[1] http://spark.apache.org/

properties of a target program. In previous work (e.g. [9]), we have used the term 'Automatic Improvement Programming' (AIP) to refer to methods which (at least in part) achieve this via semantics-preserving and/or deterministic methods. The term 'automatic' was chosen for its historical association with different methods of nonstochastic program transformation in the field of Automatic Programming [10]. In contrast, previous work using the term 'Genetic Improvement' (GI) has been predominantly concerned with the application of stochastic (if not necessarily genetic) search [11]. In particular, there has been historical emphasis in GI on validation of software mutants via testing, which is clearly not required in the case of semantics-preserving transformations. The definition of GI has recently been amended to be 'computational search to improve software while retaining its partial functionality'[2].

In 2011, Orlov and Sipper [12] used Genetic Programming (GP) [13] to automatically improve existing Java programs for well known GP benchmarks such as symbolic regression and the 'artificial ant' problem. Their approach evolved Java bytecode rather than source code directly and used the notion of 'compatible crossover' to ensure program correctness. More recently, Kocsis and Swan [14] made use of the well-known Curry-Howard isomorphism to demonstrate that it is sometimes possible to replace stochastic mutation operators with transformations obtained by deterministic proof search. Kocsis et al. [9] also describe an AIP system for repairing and improving the implementation of hashCode methods in Hadoop, an open- Java-based framework for distributed Big Data storage and processing. The Hadoop code base was analyzed and it was procedurally determined that there were over 400 cases where the hashCode implementation did not meet its contractual obligations. Semantics-preserving transformations were then performed to ensure the required contractual consistency with the equals method. Genetic Programming was then used to improve the distribution of the repaired hashCode method. The automatically improved hashCode implementations were able to outperform both the original Hadoop implementations and an existing hashcode generation tool on a number of case studies. In recent work by Burles et al. [15], the behavioral semantics of Object-Orientation is used to constrain possible substitutions for Google Guava collection classes. By employing these constraints within a genetic search, a 200-fold improvement over an exact approach is achieved for the minimization of energy consumption.

2.1 Reflection in Scala

The Hylas framework described here is implemented via the reflection capabilities of the Scala language, where reflection is the ability of a program to inspect and modify it's own behaviour. In general, reflection on abstract syntax trees (ASTs) in Scala can either be applied at compile-time or run-time (i.e. offline or online improvement respectively).

The historic trend in GI has been offline, but there is an increasing interest in online improvement of software [16, 17]. Swan et al. introduced the Gen-O-

[2] http://geneticimprovementofsoftware.com/

Fix [18] framework as an embeddable framework supporting dynamic adaptation. Gen-O-Fix uses runtime reflection to perform online transformations of Scala ASTs. If a modified AST yields better performance than the original, the mutant source and object code replaces that of the original system. This functionality is of particular value to 'always-on' systems such as web servers and embedded systems, where the mutants can be evaluated concurrently with the execution of the live system.

The compile-time transformation of ASTs in Hylas is achieved via *macros*, which differ from the notion of macros familiar to 'C' programmers in that they can make use of all available type and scope information [19]. A macro is a Scala function invoked by the compiler that directly transforms the AST of Scala source code and can therefore be considered as an open-ended extension of the compilation phase. Other work in this area that can be considered to augment compiler capabilities includes that of White [20], who investigated the ability of GP to provide a 'gracefully degraded' tradeoff between functional and non-functional properties in low-resource systems. Work in compiler augmentation more closely related to this article is that of Alexander and Gratton [21], who use Grammatical Evolution to evolve Stratego [22] rewrite rules for Data Movement Optimisation.

3 Hylas

We now describe the Hylas tool in more detail. As mentioned above, the optimising transformations applied by Hylas are taken from deforestation techniques originated by Wadler [8]. The motivation for deforestation arises from the prevalence of intermediate data structures in functional programming. For example, when using the 'pointfree' style of programming to calculate the sum of squares for the first n integers (1 **until** n).**map**(square).reduce(_ + _) requires a list to be generated at each function application. Wadler presents seven deforestation rules for functional programming [8] which can eliminate such intermediate expression in a variety of cases.

Hylas is currently able to apply simple deforestation rules of the forms given in Listing 1.1. Note that the rules map transformation-action pairs to transformation-action pairs, and can thus be applied recursively. The rules preserve query semantics. Each deforestation rule defined below eliminates intermediate data structures: RDDs that are created during execution to compute a value and are subsequently discarded.

Listing 1.1. Hylas deforestation rules

```
(rdd filter f) foreach (x => g(x))
  ->  rdd.foreach(x => if(f(x)) g(x))
(rdd map f) foreach (x => g(x))
  ->  rdd.foreach(x => g(f(x)))
(rdd map f) map g
  -> rdd map (x => g(f(x)))
```

```
(rdd map f).count
  -> rdd.count
(rdd1 cartesian rdd2).count
  -> rdd1.count * rdd2.count
```

We now explain the operation of each rule in more detail:

1. (rdd **filter** f) foreach (x =>g(x))
   → rdd foreach (x =>**if**(f(x)) g(x)):
   The original code creates an intermediate data structure, containing all the values satisfying some given predicate f, then performs the given action g on each value in the intermediate data structure. The transformed code walks the RDD once, executing the action g only on the values that satisfy f.
2. (rdd **map** f) foreach (x =>g(x))
   → rdd.foreach(x =>g(f(x))):
   The original code creates an intermediate data structure by applying the function f to each value of the RDD, then performs the given action g on each value in the intermediate data structure. The transformed code walks the RDD once, executing the action g directly on the value f(x), without creating intermediate structures.
3. (rdd **map** f) **map** g
   → rdd **map** (x =>g(f(x))):
   The original code creates an intermediate data structure by applying the function f to each value of the RDD, then applies the function g on each value in the intermediate data structure. The transformed code applies the function $f \circ g$ directly on the values in the RDD, without creating the intermediate structure.
4. (rdd **map** f).count
   → rdd.count:
   The original code creates an intermediate data structure by applying the function f to each value of the RDD, then counts the number of values in the intermediate structure. Since applying a function cannot change the number of values in the RDD, the transformed code simply returns the number of elements in the RDD, without applying the function to create the intermediate structure.
5. (rdd1 cartesian rdd2).count
   → rdd1.count * rdd2.count:
   The original code calculates the Cartesian product of two RDDs, then counts the elements of the product. The transformed code counts the elements in the product directly, without constructing the intermediate data structure.

The semantics of Spark ensure that the functions that are removed by the transformation in Rules 4 and 5 have no side effects.

3.1 Implementation

The end-user demarcates the subsystem to which Hylas is to be applied by adding the annotation @hylas to a header (be it an object, class or method

header) or an individual query. The HYLAS tool then uses Scala's pattern matching facility to find any subtree of the AST which corresponds to the above rules and applies the associated transformations. Specifically, HYLAS executes the following algorithm:

1. Walk the AST of the annotated object, identifying optimisable queries using the available type information.
2. Apply the deforestation rules one-by-one on the identified queries.
3. Return the "deforested" AST to the compiler.

Note that, apart from this single annotation, HYLAS performs without further need of human input. Here we note that as Hylas works with the Scala compiler directly, all of the computational overhead is at compile-time. On the examples tested, the execution of HYLAS is nominal when added to the execution time taken to run each query.

4 Performance

The application domain for HYLAS was provided by Keysight Technologies, the sponsors for development. Keysight builds electronic measurement and design automation solutions for both frequency and time domains which produce high speed data streams that must be processed and analysed in real-time. The instrumentation uses a common set of measurement algorithms that may be deployed to handheld, benchtop, modular or cloud targets. The aspect of GI for Keysight is the ability to adapt and deploy to multiple targets whilst maintaining integrity of measurement. The evaluation context chosen for HYLAS was the forensic analysis of network data for cybersecurity. The line rates associated with network perimeter points are multi-gigabit so the packet traces are large and are typically analysed in a cloud environment. The analysis process is typified by repeated and increasingly complex queries on a large dataset, with the goal of eliminating false positives in order to focus on the source of malicious activity.

The test data used in the performance evaluation originated from the 2013 "Infection Discovery using DNS Data" challenge of the Los Alamos National Laboratory [23]. The data consists of several months of DNS server logs, parsed into human-readable text using the `dns_parse` tool

For performance evaluation purposes, 27 gigabytes of logs were uploaded to Amazon's S3 [24] scalable cloud storage service. Given the security-sensitive nature of the application, genuine query data was unavailable and so testing was performed on 100 different Spark queries generated synthetically by building a chain of RDD transformations and actions. HYLAS was executed on the resulting queries, the measured execution times, given in seconds, can be seen in Figure 1.

It can be seen from Figure 1 that the majority of queries were not significantly affected by HYLAS, while 10 queries saw significant and highly improved execution times (significance determined via the Mann-Whitney U test with $p = 0.05$). All five of the defined deforestation rules applied at least once in a query that was statistically significantly improved. Of the 10 queries where a significant

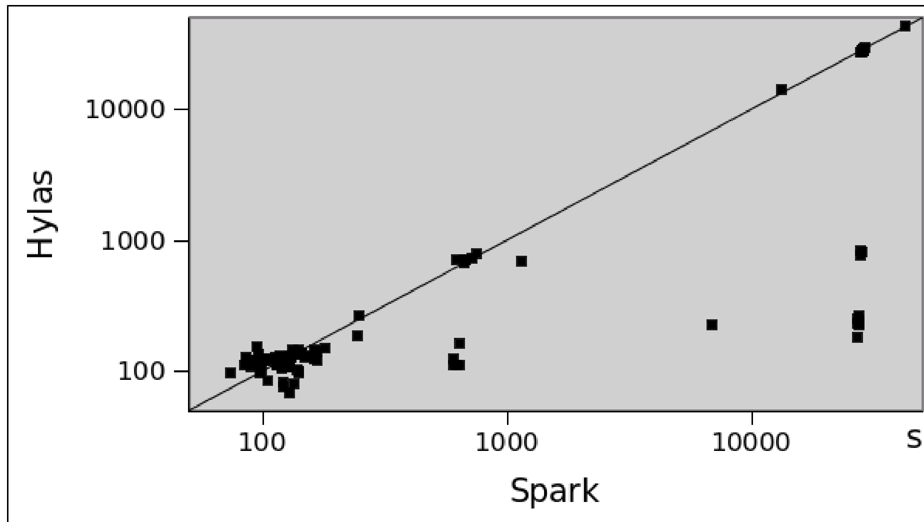

Fig. 1. Spark vs HYLAS execution times (seconds) for all queries

improvement was made, this was often due to the application of deforestation rules 3, 4 and 5 given in Section 3 where sequences of maps and Cartesian products that don't have to be constructed are present within a query. An example query from those tested is given below, where 'linear' denotes a linear time string transformation, 'quadratic' a quadratic time string transformation, and 'intlinear' and 'intquadratic' denote integer-to-integer functions of the corresponding asymptotic complexity.

```
logData1.map(quadratic).map(linear).map(x => x.length).
    map(intlinear).cartesian(logData2).count
```

Transformed to:

```
return logData1.count * logData2.count
```

For the cases where deforestation rules 1 and 2 were used, the improvements are not as high but still significant (ranging from 14 times to 180 times relative improvement). An example query and the transformed version using these two rules is given below:

```
logData1.cartesian(logData2).map(x => x.toString).map(
    linear).filter(x => x.length > 80).count
```

Transformed to:

```
val acc = accumulator(0, "counter") logData1.cartesian(
    logData2).foreach(x => if (linear(x.toString).length >
     80) acc += 1) return acc.value
```

Table 1 provides descriptive statistics of the run times of the original Spark and the HYLAS optimised variants, with the corresponding boxplots shown in Figure. 2. The average improvement was around 993 times, and in 7 of the 10 cases of significant improvement, the improvement was over 10, 000-fold. The average execution time over the 100 examples tested is also much improved when using HYLAS, taking 4,632.75 seconds compared to 7,572.06 seconds for the original Spark queries. There is little difference between the best and worst case queries in terms of execution time, in these cases it is likely that there is no difference between the original queries before and after attempting to apply the deforestation rules.

Table 1. Descriptive statistics for execution time (seconds)

	Spark	HYLAS
min	74.00	**68.00**
25%tile	118.75	**112.00**
median	143.50	**128.50**
75%ile	16,800.00	**673.75**
max	42,900.00	**42,500.00**
average	7,572.06	**4,632.75**
st. dev.	12,500.00	**10,400.00**

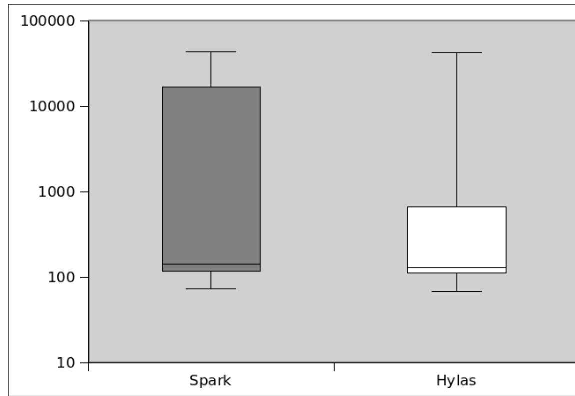

Fig. 2. Boxplot for Spark vs HYLAS execution times given in seconds

5 Conclusion and Future Work

In this paper we have described HYLAS, a tool for optimising Apache Spark queries through reflection in Scala. Using a set of rules for deforestation taken

from the functional programming literature, a set of semantics-preserving transformations are applied to each query to eliminate redundant data structures and improve efficiency. The performance evaluation shows that this approach can significantly improve execution times of some queries without adding significant compile-time overhead. Unlike many existing approaches to program improvement, HYLAS works automatically, requiring only that the end-user demarcates the subsystem to be improved by adding a single annotation to the source code. A possible limitation of this approach is that although the rules are deterministic, there may be some bias introduced by the order in which the rules are applied.

Future work will focus on extending the set of available deforestation rules. One possible approach is that of 'HFusion' [25], which uses the category-theoretic machinery of hylomorphisms to automatically deforest Haskell programs. If the increased set of deforestation rules included program transformations which conflict with one another, it would become difficult to exhaustively search all combinations of program transformations, resulting in a more traditional search problem as tackled by many existing Genetic Improvement techniques.

6 Acknowledgements

This work was funded by Keysight Technologies.

References

1. Jacques Bughin, Michael Chui, and James Manyika. Clouds, big data, and smart assets: Ten tech-enabled business trends to watch. *McKinsey Quarterly*, 56(1):75–86, 2010.
2. Marcos D Assunção, Rodrigo N Calheiros, Silvia Bianchi, Marco AS Netto, and Rajkumar Buyya. Big data computing and clouds: Trends and future directions. *Journal of Parallel and Distributed Computing*, 79:3–15, 2015.
3. Martin Odersky, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, and et al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.
4. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.
5. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
6. Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic optimization for MapReduce programs. *Proc. VLDB Endow.*, 4(6):385–396, March 2011.
7. Michael Armbrust. Catalyst - a query optimization framework for spark and shark. Presentation, Spark Summit 2013, dec 2013.

8. Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming*, pages 231–248, Amsterdam, The Netherlands, The Netherlands, 1988. North-Holland Publishing Co.
9. Zoltan A Kocsis, Geoff Neumann, Jerry Swan, Michael G Epitropakis, Alexander EI Brownlee, Sami O Haraldsson, and Edward Bowles. Repairing and optimizing hadoop hashcode implementations. In *Proceedings of the 6th International Symposium on Search-Based Software Engineering (SSBSE2015)*, volume 8636 of *LNCS*, pages 259–264. Springer, 2014.
10. A Biermann. *Encyclopedia of Artificial Intelligence, 2nd Edition*, chapter Automatic Programming, pages 18–35. John Wiley & Sons, NY, USA, 1992.
11. William B Langdon and Mark Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135, 2015.
12. Michael Orlov and Moshe Sipper. Flight of the FINCH through the Java wilderness. *Evolutionary Computation, IEEE Transactions on*, 15(2):166–182, 2011.
13. John R. Koza. *Genetic programming: on the programming of computers by means of natural selection.* The MIT Press, Cambridge, MA, 1992.
14. Zoltan A. Kocsis and Jerry Swan. Asymptotic genetic improvement programming via type functors and catamorphisms. In Colin Johnson, Krzysztof Krawiec, Alberto Moraglio, and Michael O'Neill, editors, *Semantic Methods in Genetic Programming*, Ljubljana, Slovenia, 13 September 2014. Workshop at Parallel Problem Solving from Nature 2014 conference.
15. Nathan Burles, Edward Bowles, Alexander E. I. Brownlee, Zoltan A. Kocsis, Jerry Swan, and Nadarajen Veerapen. Object-oriented genetic improvement for improved energy consumption in Google Guava. In Yvan Labiche and Marcio Barros, editors, *SSBSE*, volume 9275 of *LNCS*, pages 255–261, Bergamo, Italy, September 5-7 2015. Springer.
16. Nathan Burles, Jerry Swan, Edward Bowles, Alexander E. I. Brownlee, Zoltan A. Kocsis, and Nadarajen Veerapen. Embedded dynamic improvement. In William B. Langdon, Justyna Petke, and David R. White, editors, *Genetic Improvement 2015 Workshop*, pages 831–832, Madrid, 11-15 July 2015. ACM.
17. Kwaku Yeboah-Antwi and Benoit Baudry. Embedding adaptivity in software systems using the ECSELR framework. In *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference*, pages 839–844. ACM, 2015.
18. Jerry Swan, Michael G. Epitropakis, and John R. Woodward. Gen-O-Fix: An embeddable framework for dynamic adaptive genetic improvement programming. Technical Report CSM-195, Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA, Scotland, January 2014.
19. 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 '13, pages 3:1–3:10, New York, NY, USA, 2013. ACM.
20. David R White, Andrea Arcuri, and John A Clark. Evolutionary improvement of programs. *Evolutionary Computation, IEEE Transactions on*, 15(4):515–538, 2011.
21. B. J. Alexander and M. J. Gratton. Constructing an optimisation phase using grammatical evolution. In *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, pages 1209–1216, May 2009.
22. Eelco Visser. *Rewriting Techniques and Applications: 12th International Conference, RTA 2001 Utrecht, The Netherlands, May 22–24, 2001 Proceedings*, chapter

Stratego: A Language for Program Transformation Based on Rewriting Strategies System Description of Stratego 0.5, pages 357–361. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.
23. Paul S. Ferrell. *APT INFECTION DISCOVERY USING DNS DATA*. Apr 2013. LA-UR-13-23109,http://www.osti.gov/scitech/servlets/purl/1077023.
24. Amazon Simple Storage Service (amazon s3). Online - http://aws.amazon.com/s3/. Accessed: 08.03.2016.
25. Facundo Domínguez and Alberto Pardo. Program fusion with paramorphisms. In *Proceedings of the 2006 International Conference on Mathematically Structured Functional Programming*, MSFP'06, pages 6–6, Swinton, UK, UK, 2006. British Computer Society.

View publication stats