Extensible and Efficient Streaming Libraries

Aggelos Biboudis*

National and Kapodistrian University of Athens
Department of Informatics and Telecommunications
biboudis@di.uoa.gr

Abstract. Stream processing is mainstream (again): Widely-used stream libraries are now available for virtually all modern OO and functional languages, from Java to C# to Scala to OCaml to Haskell. Yet expressivity and performance are still lacking. This dissertation identifies the key high-level differences between various implementations, observes that future use cases are tied with past design decisions, and shows simple abstraction mechanisms are not sufficient. Is it possible to modularize the implementation of streams to enhance such libraries in terms of extensibility and performance? We present a twofold modularization of streams. To begin with, we untangle streams from the definition of their syntax and semantics and afterwards we liberate them from the need of a "sufficiently-smart" compiler. The utmost goal of this dissertation is to make streams extensible and performant, while maintaining their high level structure.
Our contributions are preceded by a performance assessment, of the current state-of-the-art of streaming libraries. Subsequently, we first propose a mechanism to enhance the maintainability of streams, supporting a high-level of extensibility. We treat streams as a domain-specific language and we design and implement **StreamAlg**, a library that has the ability to accept new operators and semantics á la carte. Next, we port the library design we used for streams to Java itself, with a lightweight tool named **Recaf**. We show how to create dialects in Java, override its semantics, support new syntactic elements and much more. Among many examples and case studies we build an extension of Java with a keyword that enables us to construct streams similar to C#. The culmination of our work is a library design, **Strymonas**, for very efficient streams while preserving their high-level nature. It explicitly avoids the reliance on black-box optimizers and "sufficiently-smart" compilers, offering highest, guaranteed and portable performance. Our approach relies on high-level concepts that are then readily mapped into an implementation.

Keywords: Code generation, domain-specific languages, multi-stage programming, optimization, stream fusion, streams

1 Introduction

Programming languages have started shifting away from the sequential programming model that the von Neumann architecture so vigorously imposed [2]. Instead

* Dissertation Advisor: Yannis Smaragdakis, Professor

of thinking in terms of *commands* and static *storage*, modern programming needs often encourage thinking in terms of processes and transformations over *flows* of data. That transition happened over several decades of research and development in programming languages, systems, and computer architectures. *Streaming* functionality is a prominent representative of this trend. Casually speaking, in computer science, a stream is a sequence of elements that can be piped through a series of transformation steps. A streaming library is a software library to manipulate streams. All streaming libraries seem to fulfill similar goals; however, their vastly different characteristics make them one of the most fascinating areas of software construction.

This dissertation investigates the modern design decisions behind the streaming libraries that are used in general-purpose programming. We identify the key high-level differences between various implementations and observe that future use cases are tied with past design decisions and simple abstraction mechanisms are not sufficient. Is it possible to modularize the implementation of streams to enhance such libraries in terms of extensibility and performance? We present a twofold modularization of streams. Firstly, we untangle streams from the definition of their syntax and semantics, and secondly, we liberate them from the need of a "sufficiently-smart" compiler. The utmost goal of this dissertation is to make streams extensible and performant, while maintaining their high level structure.

Nowadays, streaming libraries let us model algorithms as if data were in motion and not stationary: stock ticks, tweets, sales, products, inventory and real-time analytics are only some of the examples that generate petabytes of information available to data scientists. In terms of conceptual modeling, a stream corresponds to a pipe transporting gas or liquids over long distances. Materials are being processed in location A where an activity f takes place. After processing ends, each element is put in the pipe and is transferred to another location B, where f' takes place. The pipe represents the flow of data and the activities f and f' represent transformations on each element of the stream. We have only declared *what* activities take place and not *how* each transformation works. "Stream processing lets us model systems that have state without ever using assignment or mutable data" per the authors of *Structure and Interpretation of Computer Programs* [1].

A streaming library is typically offered with a set of operators to create streams, transform and consume them into scalar or other kinds of data structures, as shown in Figure 1. Its distinguishing feature in relation to simple collection APIs is that intermediate transformations are performed *on-demand*, thus they do not perform more computation than needed. Producer operators can be either backed by an in-memory data structure or not. `of_arr` creates a stream out of an array and `unfold` builds a (possibly unbounded) stream from a seed value (it unfolds a whole stream from a single value). Next are operators that transform a stream. A stream can be transformed either in a linear or a non-linear way. `map` applies a function f to each element of the input stream and returns a transformed stream. The number of elements on the input streams is equal to the number on the transformed stream. This is where linearity comes from. On the contrary,

```
// Producers of finite
val of_arr   : 'a array → 'a stream

// Producers of possibly infinite
val unfold   : ('state  → ('a * 'state) option) → 'state → 'a stream

// Transformers of linear nature
val map      : ('a → 'b) → 'a stream → 'b stream

// Transformers of non-linear nature
val filter   : ('a → bool) → 'a stream → 'a stream
val take     : int → 'a stream → 'a stream
val flat_map : ('a → 'b stream) → 'a stream → 'b stream

// Transformers of parallel loops
val zip_with : ('a → 'b  → 'c) → 'a stream → 'b stream  → 'c stream

// Consumers
val fold     : ('state → 'a → 'state) → 'state → 'a stream → 'state
```

Fig. 1: Stream Operators

`filter` applies a predicate to the input stream, again element-wise and unless the predicate is satisfied, the element does not appear on the output stream. Other operators like `take` sub-range the input stream based on a counter value. `flat_map` applies a function to each element; the results of the function application are concatenated to form the output stream, which can be a stream of zero, one or more elements. `zip_with` merges two streams according to a zipping function, applied element-wise over two streams. As expected, `zip_with` can have variations on the number of input streams as well as a default behavior like zipping two elements into a pair (called simply `zip`). Finally, we have consumers, like `fold` which apply a binary function, combining all elements of the stream. `fold` is a standard recursion operator for processing lists and can be used to fold a stream (like folding a piece of paper) into something else: `sum`, `product`, `max`, `min`, `count`, boolean operations like disjunction `or` and conjunction `and`, `concat` are only some functions that can be implemented in terms of `fold`.[1]

We present the same pipeline in two language (Figures 2 and 3). Both pipelines calculate the sum of squared elements of an array.

[1] In fact, fold is highly powerful and standard operators like `map` and `filter` can also be implemented in terms of it.

```scala
def sumOfSquares(arr : Array[Double]) : Double = {
  val sum : Double = arr.view
     .map(a_i => a_i * a_i)
     .sum
  sum
}
```

Fig. 2: Sum of squares in Scala

```java
public double sumOfSquares(double[] arr) {
  double sum = DoubleStream.of(arr)
     .map(a_i → a_i * a_i)
     .sum();
  return sum;
}
```

Fig. 3: Sum of squares in Java 8

2 Two Modern Needs: Extensibility & Performance

The rationale behind streams for general-purpose programming is that they can be used for fast data processing by providing a minimal and easy-to-use abstraction. However, the design decisions behind them tie the implementation with future use cases. Consider the mainstream, VM-based, multi-paradigm programming languages C# (through the `System.Collections.IEnumerable` interface) and Java (through the `java.util.stream` interface), which offer vastly different designs for streams. While the first offers a `zip` operator, the second does not, sacrificing the functionality in favor of performance. Another example is that Java 8 Streams, due to their internal structure, following a *push-based* design, significantly outperform C#, following a *pull-based* design, in a number of occasions. On the flipside, C# guarantees laziness in more cases, often permitting higher memory efficiency.

Two key observations motivate our study. The first is that streams need not be tightly coupled to either their implementation or the range of operators they support. The user can freely change the underlying semantics for any reason. To achieve this we propose a new design for streams, **StreamAlg**, and we view the API of streaming libraries as a domain specific language (DSL). Using that perspective we can study both their syntactic and their semantic elements. In order to modularize streams on both, we isolate the functionally-inspired API that all stream APIs share, and we propose an extensible design. This will give users the opportunity to use different flavors of streams at will. One flavor could boost performance, another could trace execution steps, yet another could be the combination of the two!

Next, we apply the same design on the programming language Java. We propose **Recaf**, a compiler that liberates both the syntax and the semantics of Java and offers the same level of extensibility at the language level. Using that compiler we are able to create extensions for constructs that do not exist in Java such as a `yield` keyword to implement iterators and subsequently a stream library that follows the C# architecture in Java.

The second observation is that modern libraries rely either on extensible compilers or on a "sufficiently-smart" dynamic compiler to generate efficient machine-level code—as if the original source code had been loop-based, hand-written code with state, mutation and ... human intuition. In the first case, for example, Haskell provides rewrite rules on the `GHC.Base` and `GHC.List` modules to perform elimination of intermediate data structures. The rules are applied at compile time [7,19]. Library authors following this strategy usually maintain two code bases (possibly in the same compilation unit; yet programming two different things): a) the library itself (following a certain pattern), and b) the optimizations in the form of rewriting rules. For the second case, of a "sufficiently-smart" dynamic compiler, the underlying VM technologies are exceptional pieces of engineering and tremendously complex, like the Java Hotspot Server Compiler [14]. However, sometimes it is difficult to predict their behavior. For example the point that a streams is used may fail to inline (unfold its body) so the quality of the expected loop can be very poor.[2] In this dissertation we view these optimizations as domain–specific entirely and implement them explicitly in the stream library itself. We propose **Strymonas**, a library that embodies the level of separability described above to streams. We implement Strymonas in both OCaml and Scala.

3 Introducing the StreamAlg design

The new design we propose offers streaming libraries *à la carte* to maximize extensibility. Our approach requires no language changes, and only leverages features found across all languages examined—i.e., standard parametric polymorphism (generics). We argue for the benefits of this design in terms of extensibility and low adoption barrier (i.e., use of only standard language features), all without sacrificing performance. Additionally, we demonstrate extensibility and provide several alternative semantics for streaming pipelines, all in an actual, publicly available implementation. Finally, we provide an example of the use of object algebras in a real-world, performance-critical setting.

Underlying our architecture is the object algebra construction of Oliveira and Cook [12] and Oliveira et al. [13]. This is combined with a library design that dissociates the push or pull nature of iteration from the operators themselves,

[2] A quote by John Rose discussing two design strategies for Java 8 Streams on the [hotspot-compiler-dev] mailing list: "HotSpot are less good at internal iterators. If the original point of the user request fails to inline all the way into the internal looping part of the algorithm (a hidden "for" loop), the quality of the loop will be very poor. "—`https://web.archive.org/web/20170322141224/http://mail.openjdk.java.net/pipermail/hotspot-compiler-dev/2015-March/017278.html`

analogously to the recent "defunctionalization of push arrays" approach in the context of Haskell [20].

In StreamAlg, a pipeline, shown earlier, gets inverted and parameterized by an `alg` object, which designates the intended semantics. For instance, a plain Java-streams-like evaluation would be written as in Figure 4.

```
PushFactory alg = new PushFactory();
int sum = Id.prj(
            alg.sum(
              alg.map(x → x * x,
                 alg.source(v))))).value;
```

Fig. 4: Example pipeline with push-based semantics

(The `Id.prj` and `value` elements in Figure 4 are part of a standard pattern for simulating higher-kinded polymorphism with plain generics. They can be ignored for the purposes of understanding our architecture.)

Although the code in Figure 4 is slightly longer than pipelines we showed earlier, its elements are highly stylized. The user can adapt the code to other pipelines with trivial effort, comparable to that of the original code fragment in Java 8 streams. Most importantly, if the user desired a different interpretation of the pipeline, the only necessary change is to the first line of the example. An interpretation that has pull semantics and fuses operators together only requires a new definition of `alg`:

```
FusedPullFactory alg = new FusedPullFactory();
... // same as earlier
```

Fig. 5: Declaration of an interpretation

Such new semantics can be defined externally to the library itself. Adding `FusedPullFactory` requires no changes to the original library code, allowing for semantics that the library designer had not foreseen.

This highly extensible design comes at no cost to performance. The new architecture introduces no extra indirection and does not prevent the JIT compiler from performing any optimization. This is remarkable, since current Java 8 streams are designed with performance in mind (cf. the earlier push-style semantics). As we show, StreamAlg matches or exceeds the performance of Java 8 streams.

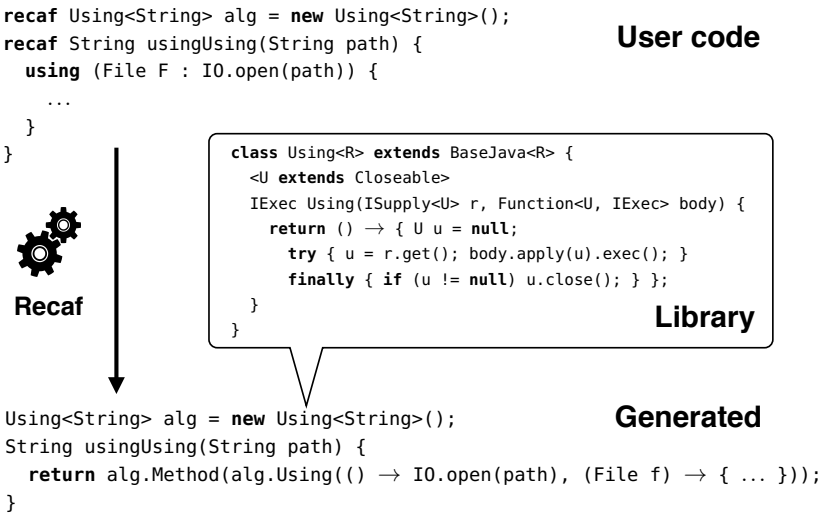

Fig. 6: High level overview of Recaf

4 Introducing the Recaf library

Figure 6 gives a bird's eye overview of Recaf. It shows how a new language extension extension is used and implemented with Recaf. The new extension offer the same functionality with `try-with-resources` in Java and is called `using`. The top shows a snippet of code illustrating how the programmer would use a Recaf extension, in this case consisting of the `using` construct. The programmer writes an ordinary method, decorated with the `recaf` modifier to trigger the source-to-source transformation. To provide the custom semantics, the user also declares a `recaf` variable, in scope of the `recaf` method. In this case, an object with static type of `Using<String>` is defined (`alg` in this example).

```
for   ( Customer c : List          ) { ... }

using ( File     f : IO.open(path) ) { ... }
```

Fig. 7: Recaf matching fragments over the concrete syntax

The downward arrow indicates Recaf's source-to-source transformation. Recaf **detects** that the new keyword relies on the `for`-each statement syntactically. An enhanced `for`-loop, in vanilla Java, omits explicit looping variables by operating

over objects of type `*Iterable`. The new keyword, `using`, relies on the same pattern and the two uses are shown below for comparison (the highlighted parts in Figure 7 show the fragments over the concrete syntax, that the Recaf compiler matches to detect the pattern).

Recaf, after detecting the concrete syntax of the pattern, virtualizes the compilation unit at the method level by transforming the code fragment that includes the extension to the plain Java code at the bottom.

Each statement in the user code is transformed into calls on the `alg` object. The `using` construct itself is mapped to the `Using` method. The `Using` class, shown in the call-out, defines the semantics for `using`. It takes two parameters: one of type `ISupply`, a lambda that takes no parameters and supplies a value (the value on the right of the semicolon) and one function of type `Function<U, IExec>` that represents the code inside the block of `using`, as a function, parameterized by a value of type `U` (one the left of the semicolon and of type `File` in this example). It extends a class (`BaseJava`) capturing the ordinary semantics of Java, and defines a single method, also called `Using`. This particular `Using` method defines the semantics of the `using` construct as a kind of interpreter, of type `IExec`.

5 Introducing the Strymonas library

We next present **Strymonas**: a streaming library design that offers both high expressiveness and *guaranteed*, highest performance. First, we support the full range of streaming operators (a.k.a. stream *transformers* or *operators*) from past libraries: not just `map` and `filter` but also sub-ranging (`take`), nesting (`flat_map`—a.k.a. `concatMap`) and parallel (`zip_with`) stream processing. All operators are freely composable: e.g., `zip_with` and `flat_map` can be used together, repeatedly, with finite or infinite streams. Our novel stream representation captures the essence of stream processing for virtually all operators examined in past literature.

Second, our stream representation allows eliminating the abstraction overhead altogether, for the full set of stream operators. We perform *stream fusion* and other aggressive optimizations. The generated code contains no extra heap allocations in the main loop. By not generating tuples or other objects, we avoid the overhead of dynamic object construction and pattern-matching, and also the hidden, often significant overhead of memory pressure and boxing of primitive types as in Java 8 (using the generic types and not the hand-specialized) and in Scala. The result not merely approaches but attains the performance of hand-optimized code, from the simplest to the most complex cases, up to *well over* the complexity point where hand-written code becomes infeasible. Although the library operators are purely functional and freely composable, the actual running stream code is loop-based, highly tangled and imperative.

Our technique relies on staging, a form of metaprogramming, to achieve guaranteed stream fusion. This is in contrast to past use of source-to-source transformations of functional languages [8], of AST run-time rewriting [11,15], compile-time macros [17] or Haskell GHC Rules [16,6] to express domain-specific streaming optimizations.

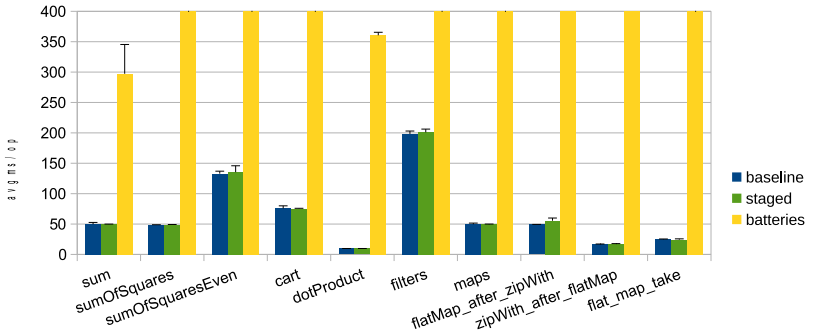

Fig. 8: OCaml microbenchmarks in msec / iteration (avg. of 30, with mean-error bars shown). "Staged" is our library (Strymonas). The figure is truncated: OCaml batteries take more than 60sec (per iteration!) for some complex benchmarks.

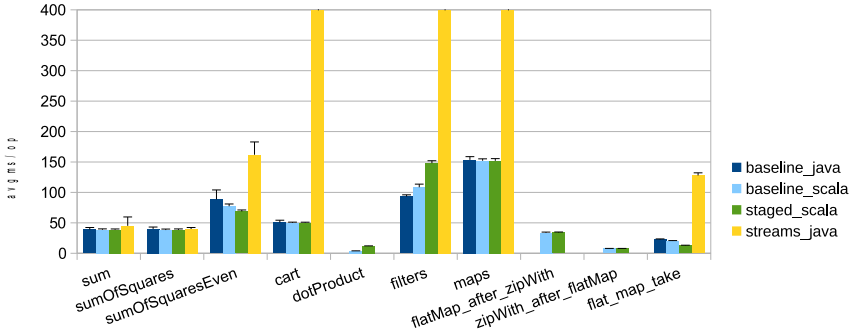

Fig. 9: JVM microbenchmarks (both Java and Scala) in msec / iteration (avg. of 30, with mean-error bars shown). "Staged_scala" is our library (Strymonas). The figure is truncated.

Rather than relying on an optimizer to eliminate artifacts of stream composition, we do not introduce the artifacts in the first place. Our library transforms highly abstract stream pipelines to code fragments that use the most suitable imperative features of the host language. The appeal of staging is its certainty and guarantees. Unlike the aforementioned techniques, staging also ensures that the generated code is well-typed and well-scoped, by construction. Our work describes a general approach, and not just a single library design. To demonstrate the generality of the principles, we implemented two library versions in diverse settings. The first is an OCaml library, staged with BER MetaOCaml [9]. The second is a Scala library (also usable by client code in Java and other JVM languages), staged with Lightweight Modular Staging (LMS) [18].

We evaluate Strymonas on a suite of benchmarks (Figures 8 and 9), comparing with hand-written code as well as with other stream libraries (including Java 8 Streams). Our staged implementation is up to more than two orders-of-magnitude faster than standard Java/Scala/OCaml stream libraries, matching

the performance of hand-optimized loops. (Indeed, we occasionally had to improve hand-written baseline code, because it was slower than the library.)

Thus, our contributions are: (i) the principles and the design of stream libraries that support the widest set of operations from past libraries and also permit the full elimination of abstraction overhead. The main principle is a novel representation of streams that captures rate properties of stream transformers and the form of termination conditions, while separating and abstracting components of the entire stream state. This decomposition of the essence of stream iteration is what allows us to perform very aggressive optimization, via staging, regardless of the streaming pipeline configuration. (ii) The implementation of the design in terms of two distinct library versions for different languages and staging methods: OCaml/MetaOCaml and Scala/JVM/LMS.

6 Conclusions

Summarizing, we improve streams in terms of extensibility and performance, and with the mechanisms we present, we enhance them without breaking their high level structure. In this dissertation we treat interpretations and optimizations as pluggable components and we advocate that domain-specific optimizations must be developed in "active" Stream APIs instead of "sufficiently-smart compilers".

7 Credits

The contents of this doctoral dissertation are based on published papers that were written in collaboration with others. Specifically:

- *Clash of the Lambdas* [5]; joint research with Nick Palladinos and Yannis Smaragdakis.
- *Streams à la carte* [4]; joint research with Nick Palladinos, George Fourtounis and Yannis Smaragdakis.
- *Recaf: Java Dialects As Libraries* [3]; work done while the author was affiliated with CWI; original design and implementation by the author, Pablo Inostroza and Tijs van der Storm; implementation of expression-level extensibility and corresponding applications by Pablo Inostroza.
- *Stream Fusion, to Completeness* [10]; original design by Oleg Kiselyov with help by the author on implementation and evaluation, jointly with Nick Palladinos and Yannis Smaragdakis.

References

1. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation of Computer Programs (1985)
2. Backus, J.: Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. Commun. ACM 21(8), 613–641 (Aug 1978)

3. Biboudis, A., Inostroza, P., Storm, T.v.d.: Recaf: Java dialects as libraries. In: Proc. of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. pp. 2–13. GPCE '16, ACM (2016)
4. Biboudis, A., Palladinos, N., Fourtounis, G., Smaragdakis, Y.: Streams à la carte: Extensible Pipelines with Object Algebras. In: Proc. of the 29th European Conference on Object-Oriented Programming. pp. 591–613. ECOOP '15, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2015)
5. Biboudis, A., Palladinos, N., Smaragdakis, Y.: Clash of the lambdas. In: Proc. 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems. ICOOOLPS '14 (2014)
6. Coutts, D., Leshchinskiy, R., Stewart, D.: Stream fusion: From lists to streams to nothing at all. In: Proc. of the 12th ACM SIGPLAN International Conference on Functional Programming. pp. 315–326. ICFP '07, ACM (2007)
7. Gill, A., Launchbury, J., Peyton Jones, S.L.: A short cut to deforestation. In: Proc. of the Conference on Functional Programming Languages and Computer Architecture. pp. 223–232. FPCA '93, ACM (1993)
8. Kelsey, R., Hudak, P.: Realistic compilation by program transformation (detailed summary). In: Proc. of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 281–292. POPL '89, ACM (1989)
9. Kiselyov, O.: The Design and Implementation of BER MetaOCaml. In: Proc. of the 12th International Symposium on Functional and Logic Programming. pp. 86–102. FLOPS '14, Springer (2014)
10. Kiselyov, O., Biboudis, A., Palladinos, N., Smaragdakis, Y.: Stream fusion, to completeness. In: Proc. of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 285–299. POPL '17, ACM (2017)
11. Murray, D.G., Isard, M., Yu, Y.: Steno: Automatic Optimization of Declarative Queries. In: Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 121–131. PLDI '11, ACM (2011)
12. Oliveira, B.C.d.S., Cook, W.R.: Extensibility for the masses: Practical extensibility with object algebras. In: Proc. of the 26th European Conference on Object-Oriented Programming, ECOOP '12, vol. 7313, pp. 2–27. Springer Berlin Heidelberg (2012)
13. Oliveira, B.C.d.S., van der Storm, T., Loh, A., Cook, W.R.: Feature–Oriented Programming with Object Algebras. In: Proc. of the 27th European Conference on Object-Oriented Programming. pp. 27–51. ECOOP '13, Springer-Verlag (2013)
14. Paleczny, M., Vick, C., Click, C.: The java hotspot^{TM} server compiler. In: Proc. of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1. pp. 1–1. JVM'01, USENIX Association (2001)
15. Palladinos, N., Rontogiannis, K.: Linqoptimizer. `https://github.com/nessos/LinqOptimizer` (2013)
16. Peyton Jones, S., Tolmach, A., Hoare, T.: Playing by the rules: Rewriting as a practical optimisation technique in GHC. In: Haskell workshop. vol. 1, pp. 203–233 (2001)
17. Prokopec, A., Petrashko, D.: ScalaBlitz: Lightning-Fast Scala collections framework. `http://scala-blitz.github.io/` (2013)
18. Rompf, T., Odersky, M.: Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In: Proc. of the 9th International Conference on Generative Programming and Component Engineering. pp. 127–136. GPCE '10, ACM (2010)
19. Svenningsson, J.: Shortcut fusion for accumulating parameters & zip-like functions. In: Proc. of the 7th ACM SIGPLAN International Conference on Functional Programming. pp. 124–132. ICFP '02, ACM (2002)

20. Svensson, B.J., Svenningsson, J.: Defunctionalizing Push Arrays. In: Proc. of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing. pp. 43–52. FHPC '14, ACM (2014)