Go Meta! A Case for Generative Programming and DSLs in Performance Critical Systems

Tiark Rompf[1], Kevin J. Brown[2], HyoukJoong Lee[2], Arvind K. Sujeeth[2], Manohar Jonnalagedda[4], Nada Amin[4], Georg Ofenbeck[5], Alen Stojanov[5], Yannis Klonatos[3], Mohammad Dashti[3], Christoph Koch[3], Markus Püschel[5], and Kunle Olukotun[2]

1 **Purdue University, USA,** {first}@purdue.edu
2 **Stanford University, USA,** {kjbrown,hyouklee,asujeeth,kunle}@stanford.edu
3 **EPFL DATA, Switzerland,** {first.last}@epfl.ch
4 **EPFL LAMP, Switzerland,** {first.last}@epfl.ch
5 **ETH Zürich, Switzerland,** {ofgeorg,astojanov,pueschel}@inf.ethz.ch

Abstract

Most performance critical software is developed using very low-level techniques. We argue that this needs to change, and that generative programming is an effective avenue to enable the use of high-level languages and programming techniques in many such circumstances.

1998 ACM Subject Classification D.1 Programming Techniques

Keywords and phrases Performance, Generative Programming, Staging, DSLs

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.238

1 The Cost of Performance

Performance critical software is almost always developed in C and sometimes even in assembly. While implementations of high-level languages have come a long way, programmers do not trust them to deliver the same reliable performance. This is bad, because low-level code in unsafe languages attracts security vulnerabilities and development is far less agile and productive. It also means that PL advances are mostly lost on programmers operating under tight performance constraints. Furthermore, in the age of heterogeneous architectures, "big data" workloads and cloud computing, a single hand-optimized C codebase no longer provides the best, or even good, performance across different target platforms with diverse programming models (multi-core, clusters, NUMA, GPU, ...).

1.1 Abstraction Without Regret

We argue for a radical rethinking of the role of high-level languages in performance critical code: developers should be able to leverage high-level programming abstractions without having to pay the hefty price in performance. The shift in perspective that enables this vision of "abstraction without regret" is a properly executed form of *generative programming*: instead of running the whole system in a high-level managed language runtime, we advocate to focus the abstraction power of high level languages on composing pieces of low-level code, making runtime code generation and domain-specific optimization a fundamental part of the program logic. This design fits naturally with a distinction into control and data paths, which already exists in many systems.

1st Summit on Advances in Programming Languages (SNAPL'15).
Eds.: Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett; pp. 238–261

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.2 Towards a Discipline of Generative Performance Programming

While the general idea of program generation is already well understood and many languages provide facilities to generate and execute code at runtime (e.g., via quotation in the LISP tradition and "eval" even in JavaScript), generative programming remains somewhat esoteric – a black art, accessible only to the most skilled and daring of programmers. We believe that generative programming should become a part of every performance-minded programmer's toolbox. What is lacking is an established discipline of practical generative performance programming, including design patterns, best practices, and well-designed teaching material. To make up for this deficiency, we advocate for a research program and education effort, and describe our ongoing work in this direction:

- Compiling queries in database systems (Section 3)
- Protocol and data format parsers (Section 4)
- Delite: A DSL compiler framework for heterogeneous hardware (Section 5)
- Spiral: Synthesis of very high-performance numeric kernels (Section 6)

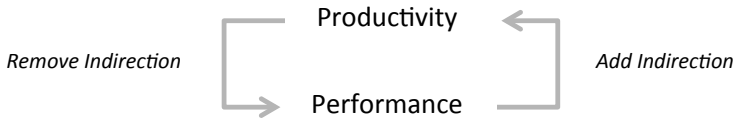
Along the way, we discuss programming patterns such as staged interpreters, mixed-stage datastructures, and how certain language features such as type classes enable powerful generative abstractions. We survey related work in Section 7, and we attempt to synthesize lessons learned and discuss limitations and challenges in Section 8.

2 Background and Context

A famous quote, attributed to David Wheeler, says that:

> "Any problem in computer science can be solved by adding a level of indirection."

What is less widely appreciated is the second part of this quote: "Except problems that are caused by too many levels of indirection". In practice, many of these problems are performance problems. More generally, there appears to be a fundamental conflict between performance and productivity: to increase productivity we need to add indirection, and to increase performance, we need to remove indirection.



2.1 About Performance

The business of program optimization is one of diminishing returns: more effort leads to increased efficiency, but further gains come at an exponentially higher price. Often, the biggest gains can be achieved by choosing a better algorithm. Thus, in the most common case, a programmer implements a work-optimal algorithm, and relies on a compiler to create an efficient mapping to architectural and microarchitectural details.

The problem with the former is that "work-optimal" is often only established asymptotically, thus ignoring constants, the choice of data structure, or locality. The problem with the latter is that compilers are automatic systems that need to work for all programs, and thus cannot be expected to deliver the best possible results for any particular program. The ability of a compiler to optimize a given program depends very much on the programming

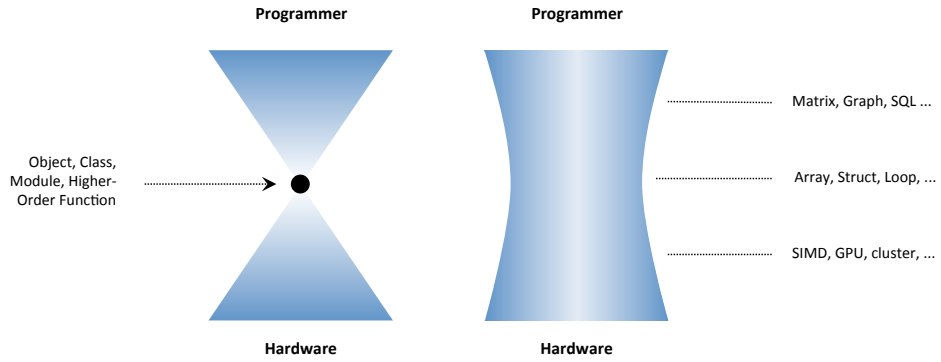


Figure 1 General purpose compilers (left) vs. DSL compiler toolchains (right).

language and on the particular programming style. In general, high-level languages are partly interpreted, with just-in-time compilers of varying sophistication. Programs in high-level languages can easily be 10x to 100x slower than equivalent C programs, and the individual coding style makes a big difference. In general, the more high-level features and indirection is used (objects, classes, higher-order functions, ...) the bigger the price in performance, because indirection fundamentally stands in the way of automatic program analysis. Thus, after algorithmic changes, a big performance boost can be obtained by rewriting a program in a low-level style, and moving "closer to the metal" by eliminating other overheads such as unnecessary data transfers. To give one example, a recent study [74] shows that simple single-threaded Rust programs can outperform big data processing frameworks running on a cluster of several hundred nodes on important graph applications. This inefficiency translates directly into higher energy consumption and data center bills.

But even reasonably optimized C implementations can be suboptimal by a large margin [12, 75], and speed-ups of sometimes 2x, 5x, or more can be achieved with careful tuning to the microarchitecture and employment of the right coding style – a non portable approach. The situation worsens considerably when parallelizing or distributing computation to homogeneous or heterogeneous processors. Here, a single C codebase is no longer sufficient as each class of devices comes with its own specific programming model (Pthreads, MPI, OpenMP, CUDA). Effectively, separate implementations are needed for each architecture, and for best performance, different heterogeneous devices may need to be combined (e.g. clusters of machines with CPUs and GPUs), which exacerbates the problem.

Compilers are unable to target this multitude of architectures automatically, mostly because they lack domain knowledge: general purpose languages provide generic abstractions such as functions, objects, etc. on top of which program-specific concepts are built, but they do not expose these more specific concepts to the compiler. Thus, the compiler is unable to restructure algorithms or rearrange data layouts to map well to a hardware platform, and cope with the large number of choices in transformations, data representations, and optimizations. This "general purpose bottleneck" is visualized in Figure 1 and contrasted with a domain-specific approach, which enables a compiler to reason about and optimize programs on multiple levels of abstraction.

2.2 About Staging

Generative or multi-stage programming (*staging* for short), goes back at least to Jørring and Scherlis [53], who observed that many programs can be separated into stages, distinguished

by frequency of execution or by availability of data. Taha and Sheard [109] made these stages explicit in the programming model and introduced MetaML as a language for multi-stage programming. A staged computation does not immediately compute a result, but it returns a program fragment that represents the computation, and that can be explicitly executed to form the next computational stage. MetaML's staging operators *bracket*, *escape* and *run* are modeled after the quote, unquote and eval facilities known from LISP and Scheme.

The presentation in this paper uses Scala and LMS (Lightweight Modular Staging) [95], a staging technique based on types. LMS is implemented as a library (hence *lightweight*), and instead of syntactic quotations, it uses Scala's overloading facilities to build staged computations. Staged expressions can be further processed and unparsed into source code in Scala or a different language, including C. Since all staged operations are defined in library code, it is easy to restrict operations to those with a C counterpart (hence *modular*). Even C constructs with no Scala counterpart (e.g. pointers) can be represented abstractly, to get the same low-level behavior and performance. The key benefit of staging is that the present-stage code can be written in a high-level style, yet generate future-stage code that is very low-level and efficient. Staging is a programmatic way to remove indirection – when generating code in one step, Scala becomes a glorified macro system to generate C code.

2.3 About DSLs

Traditionally, the appeal of DSLs is in increasing productivity by providing a higher level, more intuitive programming model for domain experts, who are not necessarily expert programmers ("user-facing" DSLs). While this is an important direction and good tool support exists (e.g. language workbenches like Spoofax [57]), our interest in DSLs is as a vehicle for exposing knowledge about high-level program structures to a compiler. The effectiveness of a DSL-based program generation approach was demonstrated, among others, by the original Spiral system, which used a complete DSL-based generative approach for the automatic parallelization and locality optimization of linear transforms [83, 84]. In such systems, DSLs are implementation details, much like intermediate representations (IRs) in a compiler, that enable specific analyses and transformations at different levels of abstractions (Figure 1).

Staging in the style of LMS is a natural fit for these "internal" DSLs: instead of generating target code directly, one simply generates expressions in a domain-specific intermediate language. The key benefit of staging remains: computation at staging time, while the DSL program is assembled, is "free", i.e. without cost at DSL runtime. Thus, the DSL does not need to include features like higher order functions, objects, etc. which are hard to analyze, but it can focus exclusively on the elements of interest (e.g. matrices, graphs, SQL queries).

The second benefit of staging in a DSL context comes in when we consider how to translate from one DSL representation to the next lower-level one. In the partial evaluation community, it is well known that specializing an interpreter yields a compiler (the first Futamura projection [35]). Thus, a staged interpreter is a translator from one language to another: in other words, we can translate from one intermediate DSL to the next by defining an interpreter for the first language, and staging it [96, 81]. This drastically reduces the effort required to implement sophisticated multi-pass compiler toolchains.

With a properly designed layered approach, building new DSLs becomes simpler as well. Instead of starting from scratch, it is sufficient to implement a DSL front-end, consisting of the required datatypes, optimizations, and translation to an already existing backend [107].

2.4 Convergence: Generative Performance Programming

DSLs and generative approaches have been around for quite some time, with a number of successes but with few examples of more mainstream adoption. SQL is perhaps the biggest DSL success story: by restricting their interface to a well defined, restricted, language, database systems are able to leverage elaborate query optimization techniques. Traditionally, database systems stop short of generating machine code to run queries, but recently query compilation is seeing a surge of interest. We will come back to this in Section 3. A recent system that is rapidly making an impact in industry is Halide [88, 87], which generates fast image processing kernels from high-level algorithmic descriptions.

We believe that generative techniques for performance optimization are an exciting area of research with the potential to deliver important tools for real world software development. The reason is in the concurrent evolution of three trends:

- **Hardware**: With contemporary and emerging architectures, the pain involved in achieving high performance has increased dramatically. Frequency scaling, and with it free speed-up, has ceased a decade ago. Systems are becoming increasingly parallel, heterogeneous, and distributed, with diverse programming models. This means that mapping algorithms optimally is more difficult than ever, and likely requires new solutions, even if domain-specific.
- **Applications**: With a shift towards big data workloads in the cloud and a proliferation of mobile devices and embedded systems, there is a growing demand for highly efficient software. Mobile users expect an "always on" experience and are growing accustomed to applications based on sophisticated machine learning algorithms that process data in near realtime. In addition to the traditional latency and throughput requirements, this demand for efficiency is increasingly driven by concerns of energy consumption, which often dominates the operating costs of a system.
- **Programming Languages**: High-level languages focus increasingly on generality and abstraction, allowing programmers to build large systems from simple but versatile parts. On the one hand, this makes it even harder for compilers to translate high-level programs to efficient code. But on the other hand, these highly expressive languages enable the sophisticated meta-programming techniques that are needed for effective generative programming.

The bottom line of our approach can be summarized succinctly as follows:

> *Use Indirection and Abstraction to Solve Performance Problems, too.*

3 Case Study: Databases and Query Processors

Popular open-source and commercial database systems have been shown [123, 104] to perform 10 to 100x worse on certain queries than specialized, hand-written C implementations of the same query. At the same time such systems contain hundreds of thousands of lines of optimized C code, with a lot of manual specialization. On last count in a recent version of the PostgreSQL server, there were about 20 distinct implementations of the memory page abstraction and 7 implementations of B-trees [61]. This form of human inlining and specialization creates a maintenance nightmare but is probably justified by improved performance.

Why, then, are database systems still falling short of hand-written queries? One reason is that most systems interpret query plans, operator by operator and record by record. Clearly

this layer of interpretation can be a bottleneck. So why aren't databases using compilers? Because compilers are way too hard to implement! In fact this is a well-known part of database folklore: The first relational DBMS, IBM's System R, initially compiled its query plans, but before the first commercial release, changed to interpretation. The reason was that code generation for the large set of query techniques being investigated was incredibly painful. Nowadays, among mainstream DBMS, only data stream processing systems such as IBM Spade or StreamBase use compilation. The reason here is that ultra-low latencies are necessary and justify the inconvenience of creating a compiler. Still, compilation has received renewed interest [60, 61, 78, 76] and very recently, compilation based systems have started to appear again (e.g. Microsoft Hekaton, Cloudera Impala and MemSQL).

The good news is that generative programming offers a generic recipe to turn interpreters into compilers: staging an interpreter enables us to specialize it with respect to any program. The result of specialization is that the interpreter is dissolved and only the computation of the interpreted program remains as residual generated code [36, 1]. We discuss two research databases systems and one tutorial system that use this technique next.

3.1 DBToaster and LegoBase

DBToaster incrementalizes query evaluators and generates low-level C++ or Scala code. The first compilation stage turns SQL queries into update event triggers. These triggers prescribe how to efficiently refresh a materialized view of the query as the base data in the database changes. The second compiler stage optimizes the event triggers and generates efficient code. Experimental results show that DBToaster improves the performance of IVM by several orders of magnitude compared to the state of the art [2]. The second-stage compiler was originally written in about 15k lines of OCAML code; recently, it was rewritten in 2k lines of Scala/LMS code. Combined with other optimizations, the query running times improved by one to two further orders of magnitude compared to the results from [2].

LegoBase (a joint project between Oracle Labs and EPFL) is an analytic query engine [59] developed from scratch using Scala and LMS. With just about 3000 lines of Scala code, it runs all 22 queries from the industry-standard TPCH benchmark, achieving up to 20x speedups over a commercial database system, and competetive performance in comparison to other state-of-the-art query compilers that are implemented using LLVM, in a much more low-level style [78]. LegoBase removes all interpretive overhead by performing whole-query optimization, and generates specialized low-level datastructures (e.g. hash tables) based on the query and data schema.

3.2 The Essence of a SQL Compiler in 500 LOC

The LegoBase design went through several iterations, and we have distilled the essence into a generative programming tutorial (presented at CUFP'15[1]). By removing functionality that was just a variation of a common theme or which did not offer specific insights (no sorting, only inner joins, ...) we were able to implement an end-to-end SQL compiler in just under 500 lines of Scala [92].

Our starting point–without any compilation–is a generic library function to read CSV files. A CSV file contains tabular data, where the first line defines the schema, i.e. column names. We would like to iterate over all rows in a file and access data fields by name:

[1] `http://scala-lms.github.io/tutorials/query.html`

```
processCSV("data.txt") { record =>            // sample data:
  if (record("Flag") == "yes")                // Name, Value, Flag
    println(record("Name"))                   // A, 7, no
}                                             // B, 2, yes
```

Records are objects of the following class, which carries both the field data and the schema, and enables lookup by key:

```
class Record(fields: Array[String], schema: Array[String]) {
  def apply(key: String) = fields(schema indexOf key)
}
```

This record class enables a nice high-level programming style but unfortunately it comes at a high price. The code above runs much slower than just writing a specialized `while` loop:

```
while (lines.hasNext) {
  val fields = lines.next().split(",")
  if (fields(2)) == yes) println(fields(0))
}
```

Being generic means that a system contains interpretive structure. In this example, we are interpreting the schema that is read from the file. In a DBMS, queries are optimized at the SQL level and then translated to low-level execution plans, which are interpreted, operator by operator.

3.2.1 Interpreter + Staging = Compiler

Let us turn our CSV reader into a query engine that can run simple SQL queries. Our example above would be expressed as

```
SELECT Name FROM data.txt WHERE Flag == 'yes'
```

and we assume that we already receive a parsed (and possibly optimized) structured representation, the execution plan. In this case:

```
Print(
  Project(List("Name"))(
    Filter(Eq(Field("Flag"),Const("yes")))(
      Scan("data.txt"))))
```

The operators are arranged in a tree, and apart from `Scan`, each of them has a parent from which it obtains a stream of records. We can implement a query interpreter as follows:

```
def eval(p: Predicate)(rec: Record): Boolean = ... // elided
def exec(o: Operator)(yld: Record => Unit) = o match {
  case Scan(file)              => processCSV(file)(yld)
  case Filter(pred)(parent)    => exec(parent) { rec => if (eval(pred)(rec)) yld(rec) }
  case Project(fields)(parent) => exec(parent) { rec => yld(fields map (k => rec(k))) }
  case Print(parent)           => exec(parent) { rec => println(rec.fields mkString ",") }
}
```

We now turn this interpreter into a compiler by adding staging using LMS (Lightweight Modular Staging) [94]. As we will see, the necessary modifications are rather minor.

3.2.2 Mixed-Stage Data Structures and Functions

LMS is a staging technique driven by types. The type `Rep[T]` represents a delayed computation of type `T`. Thus, during staging, a bare "static" type `T` means "executes now", while a wrapped "dynamic" type `Rep[T]` means "generate code to execute later". We tweak the `Record` class so that only the fields are "dynamic":

```
class Record(fields: Rep[Array[String]], schema:Array[String]) {
  def apply(key:String) = fields(schema indexOf key)
}
```

Now record objects exist purely at staging time, and never become part of the generated code. For our example, the generated code will be exactly the efficient `while` loop shown above, even if we are starting from a SQL query as our input. Let us look at the definition of procedure `processCSV`, which forms the core of our data processing engine:

```
def processCSV(file: String)(yld: Record => Rep[Unit]) = {
  val lines = FileReader(file); val schema = lines.next.split(",")
  run { while (lines.hasNext) {
      val fields = lines.next().split(",")
      yld(new Record(fields,schema))
  }}}
```

The type of the closure argument `yld` is `Record => Rep[Unit]`, a present-stage function. Invoking it will inline the computation, a key difference to a staged function of type `Rep[A=>B]`, which results in a function call in the generated code. The `while` loop is *virtualized* [93], i.e. overloaded to yield a staged expression when the condition or loop body is a `Rep` expression.

This is essentially all that is required to convert our query interpreter into a query compiler. Using types to drive staging decisions enables us to mix present-stage and future-stage code quite freely, without the syntactic noise introduced by quotation brackets. The full code adds data structure specialization, join and aggregation operators, and optimized input processing using memory-mapped IO, all in about 500 lines of Scala [92].

4 Regular Expressions and Parser Combinators

Data is not only stored but also transferred. Fast encoding and decoding of data formats is therefore a key concern in data processing pipelines. The accepted standard is to write protocol parsers by hand. Parser generators, which are common for compilers, are not frequently used. One reason is that many protocols require a level of context-sensitivity (e.g. read a value n, then read n bytes), which is not readily supported by common grammar formalisms. Many open-source projects, such as Joyent/Nginx and Apache have hand-optimized HTTP parsers, which span over 2000 lines of low-level C code. From a software engineering standpoint, big chunks of low-level code are never a desirable situation. First, there may be hidden and hard to detect security issues like buffer overflows. Second, the low-level optimization effort needs to be repeated for new protocol versions or if a variation of a protocol is desired: for example, a social network mining application may have different parsing requirements than an HTTP load balancer, although both process the same protocol.

Parser combinators could be a very attractive implementation alternative, but are usually simply too slow. Their main benefit is that the full host language can be used to compose parsers, so context sensitivity and variations of protocols are easily supported. We have recently shown how parser combinators can be implemented in a generative style [52], yielding parser generator combinators. Clever use of present-stage and staged functions enables recursive grammars and prevents code size explosion.

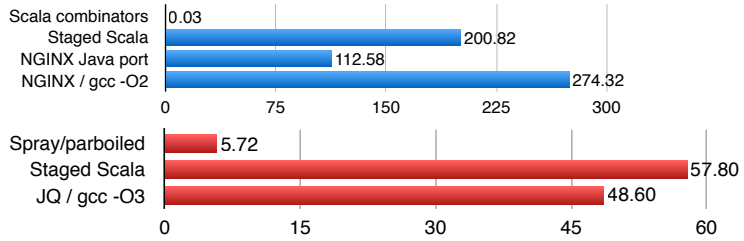


Figure 2 Parser performance (throughput in MB/s) for HTTP (top) and JSON (bottom).

We evaluate staged parser combinators against hand-written parsers for HTTP and JSON data from the NGINX and JQ projects. Our generated Scala code, running on the JVM, achieves HTTP throughput of 75% of NGINX's low-level C code, and 120% of JQ's JSON parser. Other Scala tools such as Spray are at least an order of magnitude slower. The standard Scala combinator implementation is 4 orders of magnitude slower (Figure 2).

Staged parser combinators can be implemented for different parsing styles. Communication protocols are designed to be unambiguous, so recursive decent works well and does not cause backtracking. For ambiguous grammars, we have also studied bottom up parsers [52].

An interesting special case of parsing are regular expressions. We have shown that NFA to DFA conversion can be expressed as a staged interpreter. We achieve speedups of 2x over optimized automata libraries, 100x over the JDK implementation, and more than 2000x over unstaged Scala code [96].

5 Delite: DSLs for Heterogeneous Targets

While SQL and C code generation are important use cases, contemporary "big data" workloads require more flexibility. On the application side, we need to deal with machine learning algorithms and graph processing. On the target side, we need to support GPUs, NUMA machines, and clusters. General purpose compilers are a bottleneck, unable to translate all these applications efficiently to different targets.

Delite [15, 97, 67, 68] is an extensible compiler framework for building embedded domain specific languages (DSLs). Delite supports a versatile intermediate language of parallel patterns called "Delite Ops" (including e.g. data-parallel map, filter, and reduce abstractions) to which domain-specific operations are translated, and code generation as well as runtime support for heterogeneous targets. Multiple DSLs can be used together in a single program, since all of them are translated to the common IR [107]. On the IR level, sophisticated optimizations like loop fusion and loop nesting optimizations are shared by all DSLs.

To give an example of the types of transformations that can be performed, here are two different ways of writing the computation loop (one iteration) of k-means clustering using data-parallel patterns. For shared-memory execution, the rows of the large dataset (`matrix`) are clustered, with data implicitly shuffled via the indexing operation `matrix(as)`:

```
val assigned = matrix.mapRows { row =>
  oldClusters.mapRows(centroid => dist(row, centroid)).minIndex
}
val newClusters = oldClusters.mapIndices { i =>
  val as = assigned.filter(a => a == i)
  matrix(as).sumRows.map(s => s / as.size)
}
```

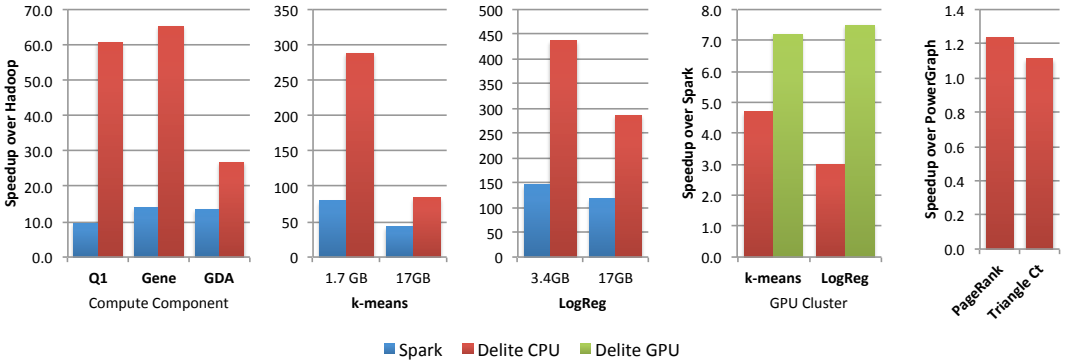


Figure 3 Delite performance for machine learning and graph apps on heterogeneous clusters.

For distributed-memory execution, however, we shuffle data explicitly via `groupBy`:

```
val clusteredData = matrix.groupRowsBy { row =>
  oldClusters.mapRows(centroid => dist(row, centroid)).minIndex
}
val newClusters = clusteredData.map(e => e.sum / e.size)
```

This way the required data movement is explicitly visible to the compiler and Delite can then reason about how to automatically partition `matrix` and the intermediate results across distributed memories and perform the required data movement. We have developed a number of DSLs for different domains, including OptiML (machine learning), OptiGraph (graph processing), OptiQL (SQL-like data querying), and OptiWrangler (data cleaning). All of these exhibit very competetive performance compared to stand-alone systems and other DSLs [106, 107]. Delite DSLs achieve high sequential performance via LMS optimizations and systematic removal of high-level domain abstractions. Parallel scalability is provided by efficient implementations of Delite Ops for multiple hardware platforms as well as the locality-improving loop transformations Delite performs on these Ops.

We show experiments comparing Delite, Spark [122], and Hadoop [11] performance on a 20 node Amazon cluster for machine learning applications, and comparing to PowerGraph [40] for graph analytics applications on a 4 node cluster connected within a single rack. The results are shown in Figure 3.

Extremely large speedups compared to Hadoop are possible for applications that perform multiple passes over a single dataset. While Hadoop loads the data from disk every iteration, Delite and Spark both keep the data in memory. Delite also shows large speedups over Spark (a Scala library) due to efficient code generation provided by LMS compared to Scala library overheads. Speedup over PowerGraph is very modest due to the graph applications being bound by network communication rather than computational efficiency.

6 Synthesis of High-Performance Numeric Kernels

For some ubiquitous computation kernels (e.g. basic linear algebra routines, FFTs, filters, Viterbi decoders), there is high demand for not only good, but absolute peak performance. The difference between a regular optimized implementation and the best known one can be staggering: for FFT kernels more than 10x, using the same algorithm and the same number of floating point operations. The difference is mostly due to low-level details such as locality optimizations, SIMD vectorization, and instruction scheduling.

```
type T = Double
add[Rep,Real,NoRep,SISD,T]
```

```
#define T double
#define N 4
void add(T* x, T* y, T* z) {
  int i = 0;
  for(; i < N; ++i) {
    T x1 = x[i];
    T y1 = y[i];
    T z1 = x1 + y1;
    z[i] = z1;
  }
}
```

(a) Staged SISD Array

```
type T = Double
add[NoRep,Real,Rep,SISD,T]
```

```
#define T double
void add(T* x, T* y, T* z) {
  T x1 = x[0]; T x2 = x[1];
  T x3 = x[2]; T x4 = x[3];
  T y1 = y[0]; T y2 = y[1];
  T y3 = y[2]; T y4 = y[3];
  T z1 = x1 + y1;
  T z2 = x2 + y2;
  T z3 = x3 + y3;
  T z4 = x4 + y4;
  z[0] = z1; z[1] = z2;
  z[2] = z3; z[3] = z4;
}
```

(b) SISD Array of Staged Doubles

```
type T = Double
add[Rep,Real,NoRep,SIMD,T]
```

```
#define T double
#define N 1
void add(T* x, T* y, T* z) {
  int i = 0;
  for(; i < N; ++i) {
    __m256d x1, y1, z1;
    x1 = _mm256_loadu_pd(x + i);
    y1 = _mm256_loadu_pd(y + i);
    z1 = _mm256_add_pd(x1, y1);
    _mm256_storeu_pd(z + i, y1);
  }
}
```

(c) Staged SIMD Array

```
type T = Double
add[NoRep,Real,Rep,SIMD,T]
```

```
#define T double
void add(T* x, T* y, T* z) {
  __m256d x1, y1, z1;
  x1 = _mm256_loadu_pd(x + 0);
  y1 = _mm256_loadu_pd(y + 0);
  z1 = _mm256_add_pd(x1, y1);
  _mm256_storeu_pd(z + 0, y1);
}
```

(d) SIMD Array of Staged Doubles

Figure 4 Different code styles generated from single vector add function (taken from [103]).

Producing optimized kernels for such numeric operations is one of the established use cases for program generation [75] with early examples including ATLAS [120], FFTW codelet generator [34], and the aforementioned Spiral [83, 84]. All three were developed from scratch as stand alone tools, which are hard to extend, reuse, combine, or maintain.

To investigate whether a) these generators can be implemented using embedded DSLs and staging, and b) what proper language support can offer for this application, we reimplemented a subset of Spiral using Scala and LMS, leading to a draft proposal for a more systematic approach to constructing this kind of program generators [81]. Spiral was chosen because it is almost completely DSL-based. In particular, Spiral uses three internal DSLs: one to capture recursive algorithms as breakdown rules, one to mathematically express loops and access patterns, and one a `C`-like intermediate representation. Algorithmic choice arises from different combinations of breakdown rules and structural optimizations for locality and parallelism are performed by rewriting on the second DSL.

The main results of this work were a) a staging approach to translate a DSL into another DSL, b) the use of type classes to abstract over data type and code style used in the generated code. We briefly survey the last contribution.

6.1 Type Classes and Generic Programming

Generated libraries often need to support multiple input/output data formats, such as interleaved, split, or C99 format of complex number arrays. A key result of the Spiral-LMS prototype has been to show that established generic programming techniques such as Haskell-style type classes [119], applied in a generative style, are well suited to model this diversity.

But type classes are even more powerful in the context of LMS, as they make it possible to abstract over staging decisions. In particular, some necessary but tedious low-level transformations such as loop unrolling with scalar replacement, selective precomputation or specialization to partially known input can all be expressed as instantiations of a generic `CVector` class with suitable type parameters.

More recent work has shown that this approach can also cover abstraction over SIMD architectures [103]. In Figure 4, we show how a generic `add` function can generate different code shapes depending on the type parameter instantiations. The definition of the `CVector` class and `add` function is given below:

```
class CVector[V[_], E[_], R[_], P[_], T](...) {
  type Element = E[R[P[T]]]
  def  size   (): V[Int]
  def  apply  (i: V[Int])  : Element
  def  update (i: V[Int], v: Element)
}
```

```
def add[V[_], E[_], R[_], P[_], T] (
  (x, y, z) : Tuple3[CVector[V,E,R,P,T]]
) = {
  for ( i <- 0 until x.size() )
    z(i) = x(i) + y(i)
}
```

- `T`: array primitive (double, float, etc).
- `P[_]`: SIMD or SISD array.
- `R[_]`: staged or non-staged elements of the array.
- `E[_]`: complex or real array elements.
- `V[_]`: encodes whether array accesses will be visible in the target code element operations.

Each type parameter, defined in the `CVector` class, is accompanied by a corresponding implicit type class instance, which defines the shape of `CVector` upon instantiation:

```
type NoRep[T] = T
class Staged_SISD_Double_Vector extends CVector[Rep  , Real, NoRep, SISD, Double]
class Scalar_SISD_Double_Vector extends CVector[NoRep, Real, Rep,   SISD, Double]
class Staged_SIMD_Double_Vector extends CVector[Rep  , Real, NoRep, SIMD, Double]
class Scalar_SIMD_Double_Vector extends CVector[NoRep, Real, Rep,   SIMD, Double]
```

The main prerogative of the `CVector` class is to generate code in lower level DSLs, `C-IR` and `I-IR`. The `C-IR` DSL represents a `C`-like intermediate language representation, and `I-IR` represents ISA-specific `C` intrinsics. Both DSLs model low level variable and array manipulations as well as arithmetic and binary operations, provided as extension to LMS.

```
val IR = new CIR (); IR.setISA(AVX2); import IR._
```

Once concrete `CVector` classes are created, the abstraction layer deals with a concrete ISA, specified in the `CIR` class, concrete staging decisions, and a concrete array type.

```
var x, y, z = new Staged_SISD_Double_Vector ()
var x, y, z = new Scalar_SISD_Double_Vector ()
```

```
var x, y, z = new Staged_SIMD_Double_Vector ()
var x, y, z = new Scalar_SIMD_Double_Vector ()
```

The `add` function operates on `CVector` elements constrained by the provided types. Type information propagates through each arithmetic operation, generating ISA-specific `C-IR` and `I-IR`. As a result, once C code is emitted, we obtain different versions (Figure 4) from a single source:

```
emitCode(add(x, y, z))
```

Abstracting over SIMD architectures enables the generator developer to deal with the higher-level specification using SIMD agnostic and architecture agnostic expressions. While SIMD-specific optimizations are required to achieve the highest performance, the abstraction layer is able to automatically invoke the particular architecture optimization, when the lower `I-IR` / `C-IR` is generated. As these optimizations are implemented in a modular fashion, they can be reused in the context of the abstraction layer, and also as stand-alone optimizations for building generators directly.

7 Related Work

Embedded DSLs. Embedded languages have a long history [64]. Hudak introduced the concept of embedding DSLs as pure libraries [45, 46]. Steele proposed the idea of "growing"

a language [102]. The concept of linguistic reuse goes back to Krishnamurthi [63]; Language virtualization to Chafi et al. [18]. The idea of representing an embedded language abstractly as methods (finally tagless) is due to Carette et al. [16] and Hofer et al. [44], going back to much earlier work by Reynolds [89]. Compiling embedded DSLs through dynamically generated ASTs was pioneered by Leijen and Meijer [69] and Elliot et al. [31]. All these works greatly inspired the development of LMS. Haskell is a popular host language for embedded DSLs [108, 39], examples being Accelerate [73], Feldspar [7], Nikola [72]. Recent work presents new approaches around quotation and normalization for DSLs [77, 22]. Other performance oriented DSLs include Firepile [80] (Scala), Terra [28, 29] (Lua). Copperhead [17] (Python). Rackets macros [112] provide full control over the syntax and semantics: Typed Racket [111] is implemented entirely as macros. DSLs are also used to target hardware generation, for example Lime [6], Chisel [8], Bluespec [5], or based on Delite/LMS [38, 37]. Related work on low-level systems oriented programming in high-level languages includes [33] and the Singularity operating system [65].

C++ Templates. Metaprogramming facilities exist in many languages, including C++ templates [113]. Expression Templates [114] can produce customized generation, and are used by Blitz++ [115]. Veldhuizen introduced active libraries [116], which are libraries that participate in compilation. Kennedy introduced telescoping languages [58], efficient DSLs created from annotated component libraries. TaskGraph [10] is a meta-programming library that sports run-time code generation in C++. Bassetti et al. review some of the challenges and benefits of expression templates[9]. Examples of successful packages are the Matrix Template Library [99], POOMA [56], ROSE [26] and the portable expression template engine PETE [25]. Many of these contain reusable generic components.

Compiler Optimizations. Compiler transformations in LMS [96] leverage work on rewrite rules [14], on combing optimizations in a modular fashion [70, 117, 23], on loop fusion [41], and on eliminating intermediate results [118, 24]. Other work focuses on tiling and loop nests[121][3]. Extensible compiler tools include Polyglot [79] and JastAdd [30]. The Glasgow Haskell Compiler also supports custom rewrite rules [51]. ROSE [86] derives source-to-source optimizations from semantics annotations on library code. COBALT [71] is a domain-specific language for optimizations amenable to automated correctness reasoning. Tate et al. [110] generate compiler optimizations automatically from program examples.

Cluster and Parallel Programming. There are many cluster programming frameworks, including MapReduce [27], Hadoop [11], Spark [122], Dryad [48]. Data parallel programming languages include NESL [13], SISAL [32], SaC [98], Fortress [55], Chapel [19], Data-Parallel Haskell [50], High Performance Fortran [43], and X10 [21]. Implicit parallelization framworks include Array Building Blocks [47], DryadLINQ [49], FlumeJava [20]. Another line of research focuses on irregular workloads [82].

Program Generators. A number of high-performance program generators have been built, for example ATLAS [120] (linear algebra), FFTW [34] (discrete fourier transform), and Spiral [85] (general linear transformations). Other systems include PetaBricks [4], CVXgen [42] and Halide [88, 87].

8 Outlook and Conclusions

In the preceding sections, we have presented several case studies of DSL-based generative performance programming with Scala and LMS. We could have mentioned more projects, e.g. generative programming for the web [62, 90] and domain-specific hardware generation [37, 38], but considered them beyond the scope of this paper.

What has enabled these results? First, we built on decades of prior art in DSLs, staging, and performance optimization, some of which we surveyed in Section 1 and Section 7. Second, all surveyed projects are collaborative efforts between experts in PL, architecture, DB, and performance optimizations. Third, the use of a common infrastructure enabled cross pollination and reuse of knowledge and code. Fourth, the embedding in Scala as host language (versus external DSLs) gave us the necessary flexibility to customize and the ability to reuse prior Java and Scala libraries. We discuss these and other important aspects below.

8.1 Modularity, Reuse, and Low Cognitive Overhead

One size does not fit all: all systems we presented use LMS with some slight variations. It is important that a system provides good core functionality out of the box but can easily accommodate extensions and modifications.

It is also important that the subjective overhead of generative programming is low, i.e. program generation should "feel" natural to programmers. While it may appear as a small detail, we have found that being able to freely mix present-stage and future-stage code with low syntactic overhead is extremely helpful. In LMS, the types in method signatures provide guidance as to which arguments are staged and which are not, as for other types in normal Scala code. As always, reading code is at least as important as writing code.

In Scala we achieve almost seamless language virtualization: allowing built-ins such as conditionals or loops to be overloaded like ordinary methods, so that they can create staged expressions. The necessary language support (Scala-Virtualized [93]) can be implemented as a modified compiler or using Scala macros.

8.2 Staging Should Preserve Semantics

Let us consider our `Record` abstraction from Section 2 with fields of type `Rep[T]`, this time using explicit quotation syntax:

```
val fields = <| lines.next().split(",") |>
yld(new Record(fields,schema))
```

If `Rep[T]` merely represents a quoted code fragment, every access to a record field may duplicate the computation! This is not only costly, but also not semantics-preserving with respect to termination and side effects. In this example:

```
processCSV("data.txt") { rec =>   <| print(${ rec("Name") }) + rec(${ "Flag") }) |>  }
```

The code fragment containing `lines.next()` would be executed twice for each record – clearly not the intended behavior. The problem is that quotation is usually a purely syntactic mechanism. To achieve semantic guarantees for realistic, call-by-value, computations we have proposed to make quotation context-sensitive [91]. We express quotation using two operators, `reflect` and `reify`, with the following reduction semantics:

```
(1)  <| foo ${ bar } baz |>          -->  reflect(" foo {" + reify { bar } + "} baz ")
(2)  reify { E[ reflect("str") ] }  -->  "val fresh = str; " + reify { E[ Code("fresh") ] }
(3)  reify { Code("str") }           --> "str"
```

Here, `E[.]` denotes a `reify`-free execution context and `fresh` a fresh identifier. As we can see, each quotation is immediately bound to a fresh identifier in the generated code, and only identifiers are passed around as `Rep` values. Thus, the evaluation order in the generated code mirrors the evaluation order of the quotations.

This development is a natural extension of previous work on quotations: Lisp introduced unhygienic `quote`/`unquote`/`eval` operators; MetaML [109] provided guarantees about the binding structure; LMS additionally maintains evaluation order.

8.3 One-Pass Code Generation is not Enough

Most previous work on staging and generative programming focused on code generation as a single pass. However, a single-pass code generator is fundamentally disadvantaged compared to a multi-pass compiler pipeline. As indicated in Section 2, low-level code is not enough to achieve best performance – for this one needs to build compiler systems around domain-specific intermediate languages. Potentially there are multiple levels of such DSLs to capture different optimization opportunities.

The key insight here is that staging greatly simplifies building such multi-pass DSL compilers: just as one can stage interpreters for regular expressions or SQL queries, many program transformations can be expressed as interpreters for intermediate languages, which are potentially annoted with auxiliary information derived by program analysis passes.

8.4 Challenges and Limitations

Understanding the limitations of a proposed techniques is key to using it effectively. First of all, we wish to note that we do not propose a silver bullet that will magically accelerate programs without help from the programmer.

Program generation shares some ideas with program synthesis, but is in a sense less ambitious. Whereas program synthesis promises to come up with an efficient and correct program automatically, generative programming still requires programming, i.e. the implementation of an algorithm. However, the tedium of manually applying performance optimizations is removed. Staging is also less ambitious than automatic just-in-time (JIT) compilation. Whereas JIT compilation aspires to identify chunks of code that are profitable to compile automatically based on runtime profiling, staging delegates such decisions to the programmer. The result is a deterministic performance model and strong guarantees about the shape of compiled code and when compilation happens.

Staging and DSLs work well if there is indeed a stage distinction: some piece of code is run much more often than another or after certain crucial data becomes available that enables optimization. A challenge in the deployment in real systems is that generated code must amortize the code generation and compilation time. Thus, staging only pays off if compiled code is used often or a single run offsets the generation cost. In Spiral, for example, code is generated offline and used many times. Many real-world systems have a natural separation into control and data paths. In these cases staging is a very good fit. In SQL databases, some queries from the TPCH benchmark take minutes to run and query compilation needs at most seconds. OptiMesh, a Delite DSL, performs iterative computations on large meshes, which would be altogether infeasible without compilation.

When going beyond single-pass code generation, a suitable DSL or potentially a stack of DSLs is needed that captures the essential degrees of freedom that can be exploited for optimization. Designing such DSLs is a creative process, and no tool can completely eliminate this effort. However, tools can help building these DSLs and implementing the

corresponding transformations. Defining a DSL in LMS requires some boilerplate, which has lead us to build Forge [105], a meta-DSL that can generate DSL implementations from declarative specifications. YinYang [54] is another approach to generate necessary definitions automatically.

Generative programming itself has a learning curve. Our experience suggests that students can be productive within a semester without prior experience with LMS. While effective generative programming demands a certain way of thinking, and teasing apart what should be static and what dynamic requires cleverness at times, we believe that generative programming is not fundamentally harder than other kinds of programming. It is easy to forget that in the early days of OOP, concepts like objects and inheritance were difficult to grasp for many programmers.

8.5 A Call to Action

As mentioned in Section 1 and throughout the paper, we argue for a rethinking of the role of high-level languages in performance critical code. In particular, as our work and the work of others have shown, generative abstractions and DSLs can help in achieving high performance by programmatically removing indirection, enabling domain-specific optimizations, and mapping code to parallel and heterogeneous platforms. Our examples used Scala, but other, similarly expressive modern languages can be used just as well, as demonstrated by Racket macros [112], DSLs Accelerate [73], Feldspar [7], Nikola [72] (Haskell), Copperhead [17] (Python), or Terra [28, 29] (Lua).

What will it take to realize the full potential of generative performance programming on a broad scale? We believe that concerted efforts along three major lines are necessary:

- **Research**: While the mechanics of code generation are well understood, we have only begun to understand what practical generative programming means. What are the programming language features that can be used to good effect in a generative style that targets performance? We have identified type classes as an important feature, others have worked with polytypic programming [100, 101] and metaobject protocols [29]. Specializing interpreters is another well known technique. What are key programming patterns beyond those?
- **Engineering**: For effective practical use, we need toolchains and frameworks that come with "batteries included", but are at the same time accessible for customization. Building and maintaining such systems takes considerable engineering effort, but pays off in the long run through reuse. A case in point is the LLVM compiler framewok [66].
- **Teaching**: Today, generative programming is somewhat of a black art. We need learning materials, textbooks, and university courses that teach effective meta-programming techniques to drive adoption beyond the perhaps 1% of programmers that are intrinsically inclined towards such techniques.

We believe that this is an exciting opportunity for PL research, with a big potential impact on the whole computing industry. If high-level programming becomes more often the tool of choice for performance critical systems, and less code is written in low-level, unsafe, languages, the overall reliability and safety of software will improve, with transitive benefits to the whole society.

Acknowledgements. The work presented in this paper has profited from numerous interactions with other researchers in the area, for example at Shonan Meetings 2012 and 2014, and annual IFIP WG 2.11 meetings. Many other people have contributed to the development of

our tools, including Nithin George, Vojin Jovanovic, Sandro Stucki, Vera Salvisberg, Adriaan Moors, Thierry Coppey, Julien Richard-Foy, Hassan Chafi, Chris De Sa, Christopher R. Aberger, Jithin Thomas, Alexey Romanov, Alexander Slesarenko, Lewis Brown.

This work is supported, among other sources, by ERC grant 587327 DOPPLER, DARPA Contract- Air Force, Xgraphs; Language and Algorithms for Heterogeneous Graph Streams, FA8750-12-2-0335; Army Contract AHPCRC W911NF-07-2-0027-1; NSF Grant, BIGDATA: Mid-Scale: DA: Collaborative Research: Genomes Galore - Core Techniques, Libraries, and Domain Specific Languages for High-Throughput DNA Sequencing, IIS-1247701; NSF Grant, SHF: Large: Domain Specific Language Infrastructure for Biological Simulation Software, CCF-1111943; Dept. of Energy- Pacific Northwest National Lab (PNNL)- Integrated Compiler and Runtime Autotuning Infrastructure for Power, Energy and Resilience-Subcontract 108845; NSF Grant- EAGER- XPS:DSD:Synthesizing Domain Specific Systems-CCF-1337375; Stanford PPL affiliates program, Pervasive Parallelism Lab.

References

1 Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, PPDP'03, pages 8–19, New York, NY, USA, 2003. ACM.

2 Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5(10):968–979, 2012.

3 Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Tiling imperfectly-nested loop nests. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 31–31. IEEE, 2000.

4 Saman P. Amarasinghe. Petabricks: a language and compiler based on autotuning. In Manolis Katevenis, Margaret Martonosi, Christos Kozyrakis, and Olivier Temam, editors, *High Performance Embedded Architectures and Compilers, 6th International Conference, HiPEAC 2011, Heraklion, Crete, Greece, January 24-26, 2011. Proceedings*, page 3. ACM, 2011.

5 Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification. In *1st ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2003), 24-26 June 2003, Mont Saint-Michel, France, Proceedings*, page 249. IEEE Computer Society, 2003.

6 Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA, pages 89–108, New York, NY, USA, 2010. ACM.

7 Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The design and implementation of feldspar: An embedded language for digital signal processing. In *Proceedings of the 22nd international conference on Implementation and application of functional languages*, IFL'10, pages 121–136, Berlin, Heidelberg, 2011. Springer-Verlag.

8 Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC'12, San Francisco, CA, USA, June 3-7, 2012*, pages 1216–1225. ACM, 2012.

9 Federico Bassetti, Kei Davis, and Daniel J. Quinlan. C++ expression templates performance issues in scientific computing. In *IPPS/SPDP*, pages 635–639, 1998.

10 Olav Beckmann, Alastair Houghton, Michael Mellor, and Paul H.J. Kelly. Runtime code generation in c++ as a foundation for domain-specific optimisation. In Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 77–210. Springer Berlin / Heidelberg, 2004.

11 Andrzej Bialecki, Michael Cafarella, Doug Cutting, and Owen O'Malley. Hadoop: a framework for running applications on large clusters built of commodity hardware. *Wiki at http://lucene. apache. org/hadoop*, 11, 2005.

12 Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *International Conference on Supercomputing (ICS)*, pages 340–347, 1997.

13 Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, 1996.

14 Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. Program transformation with scoped dynamic rewrite rules. *Fundam. Inf.*, 69:123–178, July 2005.

15 Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *PACT*, 2011.

16 Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.

17 Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP, pages 47–56, New York, NY, USA, 2011. ACM.

18 H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language Virtualization for Heterogeneous Parallel Computing. In *Onward!*, 2010.

19 Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the chapel language. *IJHPCA*, 21(3):291–312, 2007.

20 Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI. ACM, 2010.

21 Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, 2005.

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

23 Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17:181–196, March 1995.

24 Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. In *ICFP*, pages 315–326, 2007.

25 James Crotinger, Scott Haney, Stephen Smith, and Steve Karmesin. PETE: The portable expression template engine. *Dr. Dobb's J.*, October 1999.

26 Kei Davis and Daniel J. Quinlan. Rose: An optimizing transformation system for c++ array-class libraries. In Serge Demeyer and Jan Bosch, editors, *ECOOP Workshops*, volume 1543 of *Lecture Notes in Computer Science*, pages 452–453. Springer, 1998.

27 Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, OSDI, pages 137–150, 2004.

28 Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: a multi-stage language for high-performance computing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'13, Seattle, WA, USA, June 16-19, 2013*, pages 105–116, 2013.

29 Zachary DeVito, Daniel Ritchie, Matthew Fisher, Alex Aiken, and Pat Hanrahan. First-class runtime generation of high-performance types using exotypes. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 11. ACM, 2014.

30 Torbjörn Ekman and Görel Hedin. The jastadd system - modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007.

31 Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. *J. Funct. Program.*, 13(3):455–481, 2003.

32 J. McGraw et. al. SISAL: Streams and iterators in a single assignment language, language reference manual. Technical Report M-146, Lawrence Livermore National Laboratory, March 1985.

33 Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. Demystifying magic: high-level low-level programming. In Antony L. Hosking, David F. Bacon, and Orran Krieger, editors, *Proceedings of the 5th International Conference on Virtual Execution Environments, VEE 2009, Washington, DC, USA, March 11-13, 2009*, pages 81–90. ACM, 2009.

34 Matteo Frigo. A fast fourier transform compiler. In *PLDI*, pages 169–180, 1999.

35 Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

36 Yoshihiko Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.

37 Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J. Brown, Arvind K. Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*, pages 1–8. IEEE, 2014.

38 Nithin George, David Novo, Tiark Rompf, Martin Odersky, and Paolo Ienne. Making domain-specific hardware synthesis tools cost-efficient. In *2013 International Conference on Field-Programmable Technology, FPT 2013, Kyoto, Japan, December 9-11, 2013*, pages 120–127. IEEE, 2013.

39 Andy Gill. Domain-specific languages and code synthesis using haskell. *Queue*, 12(4):30:30–30:43, April 2014.

40 Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

41 Clemens Grelck, Karsten Hinckfuß, and Sven-Bodo Scholz. With-loop fusion for data locality and parallelism. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *Implementation and Application of Functional Languages*, IFL, pages 178–195. Springer Berlin / Heidelberg, 2006.

42 Martin Hanger, Tor Arne Johansen, Geir Kare Mykland, and Aage Skullestad. Dynamic model predictive control allocation using CVXGEN. In *9th IEEE International Conference on Control and Automation, ICCA 2011, Santiago, Chile, December 19-21, 2011*, pages 417–422. IEEE, 2011.

43 High Performance Fortran. `http://hpff.rice.edu/index.htm`.

44 Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of dsls. In Yannis Smaragdakis and Jeremy G. Siek, editors, *GPCE*, pages 137–148. ACM, 2008.

45 P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.

46 Paul Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, June 1998.

47 Intel. Intel array building blocks. `http://software.intel.com/en-us/articles/intel-array-building-blocks`.

48 Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys'07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys, pages 59–72, New York, NY, USA, 2007. ACM.

49 Michael Isard and Yuan Yu. Distributed data-parallel computing using a high-level programming language. In *SIGMOD'09: Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD, pages 987–994, New York, NY, USA, 2009. ACM.

50 Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *FSTTCS*, pages 383–414, 2008.

51 Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *Haskell Workshop*, 2001.

52 Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. Staged parser combinators for efficient data processing. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 637–653. ACM, 2014.

53 Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*, pages 86–96. ACM Press, 1986.

54 Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. Yin-yang: concealing the deep embedding of dsls. In Ulrik Pagh Schultz and Matthew Flatt, editors, *Generative Programming: Concepts and Experiences, GPCE'14, Vasteras, Sweden, September 15-16, 2014*, pages 73–82. ACM, 2014.

55 Guy L. Steele Jr. Parallel programming and parallel abstractions in fortress. In *IEEE PACT*, page 157, 2005.

56 Steve Karmesin, James Crotinger, Julian Cummings, Scott Haney, William Humphrey, John Reynders, Stephen Smith, and Timothy J. Williams. Array design and expression evaluation in pooma ii. In *ISCOPE*, pages 231–238, 1998.

57 Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA'10, pages 444–463, New York, NY, USA, 2010. ACM.

58 Ken Kennedy, Bradley Broom, Arun Chauhan, Rob Fowler, John Garvin, Charles Koelbel, Cheryl McCosh, and John Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(3):387–408, 2005.

59 Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

60 Christoph Koch. Abstraction without regret in data management systems. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013.

61 Christoph Koch. Abstraction without regret in database systems building: a manifesto. *IEEE Data Eng. Bull.*, 37(1):70–79, 2014.

62 Grzegorz Kossakowski, Nada Amin, Tiark Rompf, and Martin Odersky. Javascript as an embedded dsl. In *ECOOP*, pages 409–434, 2012.

63 Shriram Krishnamurthi. *Linguistic reuse*. PhD thesis, Computer Science, Rice University, Houston, 2001.

64 Peter J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

65 James R. Larus and Galen C. Hunt. The singularity system. *Commun. ACM*, 53(8):72–79, 2010.

66 C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, 2004.

67 HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, 2011.

68 HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Tiark Rompf, and Kunle Olukotun. Locality-aware mapping of nested parallel patterns on gpus. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Micro, 2014.

69 Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *DSL*, pages 109–122, 1999.

70 Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. *SIGPLAN Not.*, 37:270–282, January 2002.

71 Sorin Lerner, Todd D. Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI*, pages 220–231, 2003.

72 Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled GPU functions in Haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell'10, pages 67–78, New York, NY, USA, 2010. ACM.

73 Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional GPU programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP'13, pages 49–60, New York, NY, USA, 2013. ACM.

74 Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what cost? Technical report, Microsoft Research, 2015.

75 José M. F. Moura, Markus Püschel, David Padua, and Jack Dongarra. Scanning the issue: Special issue on program generation, optimization, and platform adaptation. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):211–215, 2005.

76 Fabian Nagel, Gavin M. Bierman, and Stratis D. Viglas. Code generation for efficient query processing in managed runtimes. *PVLDB*, 7(12):1095–1106, 2014.

77 Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. Everything old is new again: Quoted domain specific languages. Technical report, University of Edinburgh, 2015.

78 Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

79 Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *CC*, pages 138–152, 2003.

80 Nathaniel Nystrom, Derek White, and Kishen Das. Firepile: run-time compilation for GPUs in Scala. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE, pages 107–116, New York, NY, USA, 2011. ACM.

81 Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. Spiral in scala: towards the systematic construction of generators for performance libraries. In *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*, pages 125–134, 2013.

82 Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, Muhammad Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 12–25. ACM, 2011.

83 M. Püschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232 –275, feb. 2005.

84 Markus Püschel, Franz Franchetti, and Yevgen Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer, 2011.

85 Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David A. Padua, Manuela M. Veloso, and Robert W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *IJHPCA*, 18(1):21–45, 2004.

86 Daniel J. Quinlan, Markus Schordan, Qing Yi, and Andreas Sæbjørnsen. Classification and utilization of abstractions for optimization. In *ISoLA (Preliminary proceedings)*, pages 2–9, 2004.

87 Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32, 2012.

88 Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'13, Seattle, WA, USA, June 16-19, 2013*, pages 519–530, 2013.

89 J.C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. *CMU Technical Report*, 1975.

90 Julien Richard-Foy, Olivier Barais, and Jean-Marc Jézéquel. Efficient high-level abstractions for web programming. In *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*, pages 53–60, 2013.

91 Tiark Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming.* PhD thesis, EPFL, 2012.

92 Tiark Rompf and Nada Amin. A SQL to C compiler in 500 lines of code. Technical report, Purdue University, 2015.

93 Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. Scala-virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation*, pages 1–43, 2013.

94 Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE, pages 127–136, New York, NY, USA, 2010. ACM.

95 Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

96 Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs. In *POPL*, 2013.

97 Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Building-blocks for performance oriented dsls. In Olivier Danvy and Chung-chieh Shan, editors, *Proceedings IFIP Working Conference on Domain-Specific Languages, DSL 2011, Bordeaux, France, 6-8th September 2011.*, volume 66 of *EPTCS*, pages 93–117, 2011.

98 Sven-Bodo Scholz. Single assignment c: efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13(6):1005–1059, 2003.

99 Jeremy G. Siek and Andrew Lumsdaine. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, number 1505 in Lecture Notes in Computer Science, pages 59–70, 1998.

100 Alexander Slesarenko. Lightweight polytypic staging: a new approach to an implementation of nested data parallelism in scala. In *Scala Workshop*, 2012.

101 Alexander Slesarenko, Alexander Filippov, and Alexey Romanov. First-class isomorphic specialization by staged evaluation. In *Workshop on Generic Programming (WGP)*, 2014.

102 G.L. Steele. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.

103 Alen Stojanov, Georg Ofenbeck, Tiark Rompf, and Markus Püschel. Abstracting vector architectures in library generators: Case study convolution filters. In Laurie J. Hendren, Alex Rubinsteyn, Mary Sheeran, and Jan Vitek, editors, *ARRAY'14: Proceedings of the 2014 ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, Edinburgh, United Kingdom, June 12-13, 2014*, page 14. ACM, 2014.

104 Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era (it's time for a complete rewrite). In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *VLDB*, pages 1150–1160. ACM, 2007.

105 Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: generating a high performance DSL implementation from a declarative specification. In *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*, pages 145–154, 2013.

106 Arvind K. Sujeeth, HyoukJoong. Lee, Kevin J. Brown, Tiark Rompf, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning*, ICML, 2011.

107 Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksander Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In *European Conference on Object Oriented Programming*, ECOOP, 2013.

108 Bo Joel Svensson, Mary Sheeran, and Ryan Newton. Design exploration through code-generating dsls. *Queue*, 12(4):40:40–40:52, April 2014.

109 Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

110 Ross Tate, Michael Stepp, and Sorin Lerner. Generating compiler optimizations from proofs. In *POPL*, pages 389–402, 2010.

111 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In George C. Necula and Philip Wadler, editors, *POPL*, pages 395–406. ACM, 2008.

112 Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI'11, pages 132–141, New York, NY, USA, 2011. ACM.

113 D. Vandevoorde and N.M. Josuttis. *C++ templates: the Complete Guide*. Addison-Wesley Professional, 2003.

114 Todd L. Veldhuizen. Expression templates, C++ gems. SIGS Publications, Inc., New York, NY, 1996.

115 Todd L. Veldhuizen. Arrays in blitz++. In *ISCOPE*, pages 223–230, 1998.

116 Todd L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University Computer Science, May 2004.

117 Todd L. Veldhuizen and Jeremy G. Siek. Combining optimizations, combining theories. Technical report, Indiana University, 2008.

118 Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.

119 Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76, 1989.

120 R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

121 Michael E Wolf and Monica S Lam. A loop transformation theory and an algorithm to maximize parallelism. *Parallel and Distributed Systems, IEEE Transactions on*, 2(4):452–471, 1991.

122 Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael 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, 2011.

123 Marcin Zukowski, Peter A. Boncz, Niels Nes, and Sándor Héman. Monetdb/x100 - a dbms in the cpu cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.