Abstraction Without Regret for Efficient Data Processing

Tiark Rompf [‡*] Nada Amin[*] Thierry Coppey[†] Mohammad Dashti[†] Manohar Jonnalagedda[*]
Yannis Klonatos[†] Martin Odersky[*] Christoph Koch[†]

[‡]Oracle Labs: {first.last}@oracle.com [*]Programming Methods Lab / [†] DATA Lab, EPFL: {first.last}@epfl.ch

Growing data sets require efficiency on all levels of the processing stack. This leads to a trade-off between generality and specialization: On the one hand, we want reusable, generic solutions that can support many different kinds of data and many different processing tasks. But on the other hand, programs need to be specialized to data schemata and execution environments to obtain good performance. To give a real-world example, popular open-source and commercial database systems have been shown [10, 12] to perform 10 or 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, which suggests that manual optimization may not be cost effective. The database community has realized this problem, with prominent researchers arguing to replace generic database systems with specialized solutions [9].

In this talk, we make the case for a) more collaboration between DB and PL researchers, and b) for using cutting-edge PL technology such as generative metaprogramming (staging [11]) to turn interpreters, which are ubiquitous in data processing pipelines, into compilers. We present a range of examples from previous and ongoing work in the context of Scala and LMS (Lightweight Modular Staging) [8], including recent collaborative efforts on developing database systems using these techniques. As a take-away for programming language designers, we further argue that for truly expressive multi-stage programming, quotation mechanisms should offer more semantics-preservation guarantees, in particular about maintaining statement execution order across stage boundaries.

Motivating Example Let us consider a small programming example in Scala. We would like to implement a generic library function to read CSV files. A CSV file contains tabular data, where the first line defines the schema, i.e. the names of the columns. We would like to iterate over all the rows in a file and access the data fields by name:

```
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 abstraction 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.

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 with the exception of `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 ",") }
}
```

It is easy to see that this layer of interpretation can be a bottleneck, not only in our little example but also in real-world data processing systems. In fact, everybody knows that interpreted code is slower than compiled code. So why aren't we using compilers everywhere? Because they 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, this was changed to interpretation. The reason for this 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. Also, limited functionality and expressive power are deemed acceptable in data stream processors, but not in general purpose DBMS.

The good news is that staging 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 [1, 4].

Mixed-Stage Data Structures and Functions Lightweight Modular Staging (LMS) [8] 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 a staged function of type `Rep[A=>B]`, which would results in a function call in the generated code. The `while` loop is *virtualized* [7], i.e. overloaded to yield a staged expression when the condition or loop body is a `Rep` expression. Using types to drive staging decisions enables us to mix present-stage and future-stage code quite freely, without syntactic noise introduced by quotation brackets.

Preserving Semantics What level of language support is required for expressive multi-stage programming? A growing number of languages support some form of code quotation, but unfortunately there are some shortcomings to purely syntactic approaches. Apart from syntactic questions, let us consider our `Record` abstraction 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 propose to make quotation context-sensitive. We introduce reflect/reify operators, and give the following reduction semantics for quotations:

```
(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 [11] provided guarantees about the binding structure; LMS additionally maintains evaluation order.

Case Studies

DBToaster DBToaster incrementalizes query evaluators and generates low-level C++ or Scala code. In a first compilation stage, DBToaster takes an SQL query and turns it 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. Compared to classical incremental view maintenance (IVM) as implemented in most commercial DBMS, DBToaster's incrementalization technique is far more aggressive. Its output consists of more fine-grained operations than those found in classical query (or view refresh) plans; DBToaster triggers naturally lend themselves to compilation. In DBToaster's second stage, the event triggers are fed to a compiler that lowers and optimizes them 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, we rewrote it in 2k lines of Scala/LMS code. Within this much reduced codebase, we were able to perform more aggressive inlining than in the original compiler as well as data structure specialization, which is relatively easy to achieve in LMS but was beyond the scope of our original efforts. As a consequence, the running times of queries compiled using the LMS-based DBToaster improve by one to two further orders of magnitude compared to the results reported in [2], on the same benchmark.

LegoBase This joint project of Oracle Labs and EPFL attempts to prove three theses:
(1) It is possible to write the query processor of a mainstream relational DBMS in a high-level language such as Scala without regret; by optimizing compilation using LMS, we can achieve code competitive in performance with the expert-written specialized C code currently used in such query engines. Due to the high breadth and variability of this optimization domain, we have to expect that a compiler will permanently be at a disadvantage compared to a creative human; however, this is offset by the compiler's much greater capacity to specialize code in many ways. For example, 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. This form of human inlining and code specialization of course creates a code maintenance nightmare but is probably justified by improved performance. One can expect that performance can benefit from further specialization and inlining that would be just too painful for a human to carry out and take responsibility for in a software development project.
(2) We can leave it to the programmer to choose one from a number of popular architectures for building such engines (say, whether to employ a Volcano style iterator model [5] or a push model for sending partial query results through the system), and have the compiler eliminate any artifacts of such a choice, always creating the same, highly optimized code.
(3) Cross-operator optimization yields practical performance benefits. State of the art query processors have an "expressiveness bottleneck" in the query plan IR, which is the interface between query optimization and query execution. Query execution is either based on interpretation or template-expansion based optimization, neither of which can benefit from optimization across relational algebra operator boundaries. By lowering these operators of the LMS IR, further cross-operator optimization is possible, and, in fact, even performed automatically by LMS.

This project is work in progress. We have built a prototype query engine based on LMS that can process query plans exported from the optimizer of a commercial relational DBMS, and have evaluated it on the standard TPC-H benchmark. This has provided strong support for theses 1 and 3. We are currently working on extending our compiler by the code transforms to realize thesis 2.

Delite and Spiral Delite [3] is compiler and runtime framework for embedded DSLs, which compete with specialized, external DSLs and manually optimized code. Spiral-S [6] is a program generator for high-performance linear transforms. Both systems are built with LMS and use staged interpreters internally as translators from high-level to low-level representations in a multi-stage compiler pipeline.

Regular expressions and Parser Combinators We compile regular expressions matchers and more general combinator parsers for communication protocols and data formats into low-level code. For regular expressions, we show 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.

We evaluate staged parser combinators by comparing to 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 based tools such as Spray are at least an order of magnitude slower.

Relite and Lancet Staged interpreters can also be used to build just-in-time compilers. Relite and Lancet are recent projects in the context of the R language and Java bytecode [1]

References

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. 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] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5(10):968–979, 2012.

[3] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. PACT, 2011.

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

[5] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 209–218. IEEE Computer Society, 1993.

[6] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel. Spiral in scala: towards the systematic construction of generators for performance libraries. In *GPCE*, pages 125–134, 2013.

[7] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation*, pages 1–43, 2013.

[8] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

[9] M. Stonebraker and U. Çetintemel. "one size fits all": An idea whose time has come and gone (abstract). In *ICDE*, pages 2–11, 2005.

[10] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[11] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[12] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. Monetdb/x100 - a dbms in the cpu cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.

[1] `github.com/tiarkrompf/relite`,`github.com/tiarkrompf/lancet`