Functional Pearl: A SQL to C Compiler in 500 Lines of Code

Tiark Rompf [*] Nada Amin[‡]

[*]Purdue University, USA: {first}@purdue.edu
[‡]EPFL, Switzerland: {first.last}@epfl.ch

Abstract

We present the design and implementation of a SQL query processor that outperforms existing database systems and is written in just about 500 lines of Scala code – a convincing case study that high-level functional programming can handily beat C for systems-level programming where the last drop of performance matters.

The key enabler is a shift in perspective towards generative programming. The core of the query engine is an interpreter for relational algebra operations, written in Scala. Using the open-source LMS Framework (Lightweight Modular Staging), we turn this interpreter into a query compiler with very low effort. To do so, we capitalize on an old and widely known result from partial evaluation known as Futamura projections, which state that a program that can specialize an interpreter to any given input program is equivalent to a compiler.

In this pearl, we discuss LMS programming patterns such as mixed-stage data structures (e.g. data records with static schema and dynamic field components) and techniques to generate low-level C code, including specialized data structures and data loading primitives.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors–Code Generation, Optimization, Compilers; H.2.4 [*Database Management*]: Systems–Query Processing

Keywords SQL, Query Compilation, Staging, Generative Programming, Futamura Projections

1. Introduction

Let's assume we want to implement a serious, performance critical piece of system software, like a database engine that processes SQL queries. Would it be a good idea to pick a high-level language, and a mostly functional style? Most people would answer something in the range of "probably not" to "you gotta be kidding": for systems level programming, C remains the language of choice.

But let us do a quick experiment. We download a dataset from the Google Books NGram Viewer project: a 1.7 GB file in CSV format that contains book statistics of words starting with the letter 'a'. As a first step to perform further data analysis, we load this file into a database system, for example MySQL:

```
mysqlimport --local mydb 1gram_a.csv
```

When we run this command we can safely take a coffee break, as the import will take a good five minutes on a decently modern laptop. Once our data has loaded, and we have returned from the break, we would like to run a simple SQL query, perhaps to find all entries that match a given keyword:

```
select * from 1gram_a where phrase = 'Auswanderung'
```

Unfortunately, we will have to wait another 50 seconds for an answer. While we're waiting, we may start to look for alternative ways to analyze our data file. We can write an AWK script to process the CSV file directly, which will take 45 seconds to run. Implementing the same query as a Scala program will get us to 13 seconds. If we are still not satisfied and rewrite it in C using memory-mapped IO, we can get down to 3.2 seconds.

Of course, this comparison may not seem entirely fair. The database system is generic. It can run many kinds of query, possibly in parallel, and with transaction isolation. Hand-written queries run faster but they are one-off, specialized solutions, unsuited to rapid exploration. In fact, this gap between general-purpose systems and specialized solutions has been noted many times in the database community [20, 24], with prominent researchers arguing that "one size fits all" is an idea whose time has come and gone [19]. While specialization is clearly necessary for performance, wouldn't it be nice to have the best of both worlds: being able to write generic high-level code while programmatically deriving the specialized, low-level, code that is executed?

In this pearl, we show the following:

- Despite common database systems consisting of millions of lines of code, the essence of a SQL engine is nice, clean and elegantly expressed as a functional interpreter for relational algebra – at the expense of performance compared to hand written queries. We present the pieces step by step in Section 2.
- While the straightforward interpreted engine is rather slow, we show how we can turn it into a query compiler that generates fast code with very little modifications to the code. The key technique is to *stage* the interpreter using LMS (Lightweight Modular Staging [17]), which enables specializing the interpreter for any given query (Section 3).
- Implementing a fast database engine requires techniques beyond simple code generation. Efficient data structures are a key concern, and we show how we can use staging to support specialized hash tables, efficient data layouts (e.g. column storage), as well as specialized type representations and IO handling to eliminate data copying (Section 4).

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...
http://dx.doi.org/10.1145/2784731.2784760

```
tid,time,title,room
1,09:00 AM,Erlang 101 - Actor and Multi-Core Programming,New York Central
2,09:00 AM,Program Synthesis Using miniKanren,Illinois Central
3,09:00 AM,Make a game from scratch in JavaScript,Frisco/Burlington
4,09:00 AM,Intro to Cryptol and High-Assurance Crypto Engineering,Missouri
5,09:00 AM,Working With Java Virtual Machine Bytecode,Jeffersonian
6,09:00 AM,Let's build a shell!,Grand Ballroom E
7,12:00 PM,Golang Workshop,Illinois Central
8,12:00 PM,Getting Started with Elasticsearch,Frisco/Burlington
9,12:00 PM,Functional programming with Facebook React,Missouri
10,12:00 PM,Hands-on Arduino Workshop,Jeffersonian
11,12:00 PM,Intro to Modeling Worlds in Text with Inform 7,Grand Ballroom E
12,03:00 PM,Mode to Joy - Diving Deep Into Vim,Illinois Central
13,03:00 PM,Get 'go'ing with core.async,Frisco/Burlington
14,03:00 PM,What is a Reactive Architecture,Missouri
15,03:00 PM,Teaching Kids Programming with the Intentional Method,Jeffersonian
16,03:00 PM,Welcome to the wonderful world of Sound!,Grand Ballroom E
```

Figure 1. Input file `talks.csv` for running example.

The SQL engine presented here is decidedly simple. A more complete engine, able to run the full TPCH benchmark and implemented in about 3000 lines of Scala using essentially the same techniques has won a best paper award at VLDB'14 [10]. This pearl is a condensed version of a tutorial given at CUFP'14, and an attempt to distill the essence of the VLDB work. The full code accompanying this article is available online at:

`scala-lms.github.io/tutorials/query.html`

2. A SQL Interpreter, Step by Step

We start with a small data file for illustration purposes (see Figure 1). This file, `talks.csv` contains a list of talks from a recent conference, with id, time, title of the talk, and room where it takes place.

It is not hard to write a short program in Scala that processes the file and computes a simple query result. As a running example, we want to find all talks at 9am, and print out their room and title. Here is the code:

```
printf("room,title")
val in = new Scanner("talks.csv")
in.next('\n')
while (in.hasNext) {
  val tid   = in.next(',')
  val time  = in.next(',')
  val title = in.next(',')
  val room  = in.next('\n')
  if (time == "09:00 AM")
    printf("%s,%s\n",room,title)
}
in.close
```

We use a `Scanner` object from the standard library to tokenize the file into individual data fields, and print out only the records and fields we are interested in.

Running this little program produces the following result, just as expected:

```
room,title
New York Central,Erlang 101 - Actor and Multi-Core Programming
Illinois Central,Program Synthesis Using miniKanren
Frisco/Burlington,Make a game from scratch in JavaScript
Missouri,Intro to Cryptol and High-Assurance Crypto Engineering
Jeffersonian,Working With Java Virtual Machine Bytecode
Grand Ballroom E,Let's build a shell!
```

While it is relatively easy to implement very simple queries in such a way, and the resulting program will run very fast, the complexity gets out of hand very quickly. So let us go ahead and add some abstractions to make the code more general.

The first thing we add is a class to encapsulate data records:

```
case class Record(fields: Fields, schema: Schema) {
  def apply(name:  String) = fields(schema indexOf name)
  def apply(names: Schema) = names map (apply _)
}
```

And some auxiliary type definitions:

```
type Fields = Vector[String]
type Schema = Vector[String]
```

Each records contains a list of field values and a *schema*, a list of field names. With that, it provides a method to look up field values, given a field name, and another version of this method that return a list of values, given a list of names. This will make our code independent of the order of fields in the file. Another thing that is bothersome about the initial code is that I/O boilerplate such as the scanner logic is intermingled with the actual data processing. To fix this, we introduce a method `processCSV` that encapsulates the input handling:

```
def processCSV(file: String)(yld: Record => Unit): Unit = {
  val in = new Scanner(file)
  val schema = in.next('\n').split(",").toVector
  while (in.hasNext) {
    val fields = schema.map(n=>in.next(if(n==schema.last)'\n'else','))
    yld(Record(fields, schema))
  }
}
```

This method fully abstracts over all file handling and tokenization. It takes a file name as input, along with a callback that it invokes for each line in the file with a freshly created record object. The schema is read from the first line of the file.

With these abstractions in place, we can express our data processing logic in a much nicer way:

```
printf("room,title")
processCSV("talks.csv") { rec =>
  if (rec("time") == "09:00 AM")
    printf("%s,%s\n",rec("room"),rec("title"))
}
```

The output will be exactly the same as before.

Parsing SQL Queries While the programming experience has much improved, the query logic is still essentially hardcoded. What if we want to implement a system that can itself answer queries from the outside world, say, respond to SQL queries it receives over a network connection?

We will build a SQL interpreter on top of the existing abstractions next. But first we need to understand what SQL queries *mean*. We follow the standard approach in database systems of translating SQL statements to an internal *query execution plan* representation– a tree of relational algebra operators. The `Operator` data type is defined in Figure 2, and we will implement a function `parseSql` that produces instances of that type.

Here are a few examples. For a query that returns its whole input, we get a single table scan operator:

```
parseSql("select * from talks.csv")
↪ Scan("talks.csv")
```

If we select specific fields, with possible renaming, we obtain a *projection* operator with the table scan as parent:

```
parseSql("select room as where, title as what from talks.csv")
↪ Project(Vector("where","what"),Vector("room","title"),
      Scan("talks.csv"))
```

And if we add a condition, we obtain an additional *filter* operator:

```
parseSql("select room, title from talks.csv where time='09:00 AM'")
↪ Project(Vector("room","title"),Vector("room","title"),
      Filter(Eq(Field("time"),Value("09:00 AM")),
        Scan("talks.csv")))
```

```
// relational algebra ops
sealed abstract class Operator
case class Scan(name: Table) extends Operator
case class Print(parent: Operator) extends Operator
case class Project(out: Schema, in: Schema, parent: Operator) extends Operator
case class Filter(pred: Predicate, parent: Operator) extends Operator
case class Join(parent1: Operator, parent2: Operator) extends Operator
case class HashJoin(parent1: Operator, parent2: Operator) extends Operator
case class Group(keys: Schema, agg: Schema, parent: Operator) extends Operator

// filter predicates
sealed abstract class Predicate
case class Eq(a: Ref, b: Ref) extends Predicate
case class Ne(a: Ref, b: Ref) extends Predicate

sealed abstract class Ref
case class Field(name: String) extends Ref
case class Value(x: Any) extends Ref
```

Figure 2. Query plan language (relational algebra operators)

```
def stm: Parser[Operator] =
  selectClause ~ fromClause ~ whereClause ~ groupClause ^^ {
    case p ~ s ~ f ~ g => g(p(f(s))) }
def selectClause: Parser[Operator=>Operator] =
  "select" ~> ("*" ^^^ idOp | fieldList ^^ {
    case (fs,fs1) => Project(fs,fs1,_:Operator) })
def fromClause: Parser[Operator] =
  "from" ~> joinClause
def whereClause: Parser[Operator=>Operator] =
  opt("where" ~> predicate ^^ { p => Filter(p, _:Operator) })
def joinClause: Parser[Operator] =
  repsep(tableClause, "join") ^^ { _.reduce((a,b) => Join(a,b)) }
def tableClause: Parser[Operator] =
  tableIdent ^^ { case table => Scan(table, schema, delim) } |
  ("(" ~> stm <~ ")")
// 30 lines elided
```

Figure 3. Combinator parsers for SQL grammar

Finally, we can use joins, aggregations (groupBy) and nested queries. Here is a more complex query that finds all different talks that happen at the same time in the same room (hopefully there are none!):

```
parseSql("select *
  from (select time, room, title as title1 from talks.csv)
  join (select time, room, title as title2 from talks.csv)
  where title1 <> title2")
↪ Filter(Ne(Field("title1"),Field("title2")),
      Join(
        Project(Vector("time","room","title1"),Vector(...),
          Scan("talks.csv")),
        Project(Vector("time","room","title2"),Vector(...),
          Scan("talks.csv")))
```

In good functional programming style, we use Scala's combinator parser library to define our SQL parser. The details are not overly illuminating, but we show an excerpt in Figure 3. While the code may look dense on first glance, it is rather straightforward when read top to bottom. The important bit is that the result of parsing a SQL query is an `Operator` object, which we will focus on next.

Interpreting Relational Algebra Operators Given that the result of parsing a SQL statement is a query execution plan, we need to specify how to turn such a plan into actual query execution. The classical database model would be to define a stateful iterator interface with `open`, `next`, and `close` functions for each type of operator (also known as *volcano model* [7]). In contrast to this traditional pull-driven execution model, recent database work proposes a push-driven model to reduce indirection [13].

Working in a functional language, and coming from a background informed by PL theory, a push model is a more natural fit from the start: we would like to give a compositional account of what an operator does, and it is easy to describe the semantics of each operator in terms of what records it pushes to its caller. This means that we can define a semantic domain as type

```
type Semant = (Record => Unit) => Unit
```

with the idea that the argument is a callback that is invoked for each emitted record. With that, we describe the meaning of each operator through a function `execOp` with the following signature:

```
def execOp: Operator => Semant
```

Even without these considerations, we might pick the push-mode of implementation for completely pragmatic reasons: the executable code corresponds almost directly to a textbook definition of the query operators, and it would be hard to imagine an implementation that is clearer or more concise. The following code might therefore serve as a definitional interpreter in the spirit of Reynolds [14]:

```
def execOp(o: Operator)(yld: Record => Unit): Unit = o match {
  case Scan(filename) =>
    processCSV(filename)(yld)
  case Print(parent) =>
    execOp(parent) { rec =>
      printFields(rec.fields) }
  case Filter(pred, parent) =>
    execOp(parent) { rec =>
      if (evalPred(pred)(rec)) yld(rec) }
  case Project(newSchema, parentSchema, parent) =>
    execOp(parent) { rec =>
      yld(Record(rec(parentSchema), newSchema)) }
  case Join(left, right) =>
    execOp(left) { rec1 =>
      execOp(right) { rec2 =>
        val keys = rec1.schema intersect rec2.schema
        if (rec1(keys) == rec2(keys))
          yld(Record(rec1.fields ++ rec2.fields,
                     rec1.schema ++ rec2.schema)) }}

}
```

So what does each operator do? A table scan just means that we are reading an input file through our previously defined `processCSV` method. A print operator prints all the fields of every record that its parent emits. A filter operator evaluates the predicate, for each record its parents produces, and if the predicate holds it passes the record on to its own caller. A projection rearranges the fields in a record before passing it on. A join, finally, matches every single record it receives from the left against all records from the right, and if the fields with a common name also agree on the values, it emits a combined record. Of course this is not the most efficient way to implement a join, and adding an efficient hash join operator is straightforward. The same holds for the group-by operator, which we have omitted so far. We will come back to this in Section 4.

To complete this section, we show the auxiliary functions used by `execOp`:

```
def evalRef(p: Ref)(rec: Record) = p match {
  case Value(a: String) => a
  case Field(name) => rec(name)
}

def evalPred(p: Predicate)(rec: Record) = p match {
  case Eq(a,b) => evalRef(a)(rec) == evalRef(b)(rec)
  case Ne(a,b) => evalRef(a)(rec) != evalRef(b)(rec)
}

def printFields(fields: Fields) =
  printf(fields.map(_ => "%s").mkString("",",","\n"), fields: _*)
```

Finally, to put everything together, we provide a main object that integrates parsing and execution, and that can be used to run queries against CSV files from the command line:

```
object Engine {
  def main(args: Array[String]) {
    if (args.length != 1)
      return println("usage: engine <sql>")
    val ops = parseSql(args(0))
    execOp(Print(ops)) { _ => }
  }
}
```

With the code in this section, which is about 100 lines combined, we have a fully functional query engine that can execute a practically relevant subset of SQL.

But what about performance? We can run the Google Books query on the 1.7 GB data file from Section 1 for comparison, and the engine we have built will take about 45 seconds. This is about the same as an AWK script, which is also an interpreted language. Compared to our starting point, handwritten scripts that ran in 10s, the interpretive overhead we have added is clearly visible.

3. From Interpreter to Compiler

We will now show how we can turn our rather slow query interpreter into a query compiler that produces Scala or C code that is practically identical to the handwritten queries that were the starting point of our development in Section 2.

Futamura Projections The key idea behind our approach goes back to early work on partial evaluation in the 1970'ies, namely the notion of *Futamura Projections* [6]. The setting is to consider programs with two inputs, one designated as static and one as dynamic. A program specializer or partial evaluator `mix` is then able to specialize a program `p` with respect to a given static input. The key use case is if the program is an interpreter:

```
result   = interpreter(source, input)
```

Then specializing the interpreter with respect to the source program yields a program that performs the same computation on the dynamic input, but faster:

```
target   = mix(interpreter, source)
result   = target(input)
```

This application of a specialization process to an interpreter is called the first Futamura projection. In total there are three of them:

```
target   = mix(interpreter, source)    (1)
compiler = mix(mix, interpreter)       (2)
cogen    = mix(mix, mix)               (3)
```

The second one says that if we can automate the process of specializing an interpreter to any static input, we obtain a program equivalent to a compiler. Finally the third projection says that specializing a specializer with respect to itself yields a system that can generate a compiler from any interpreter given as input [3].

In our case, we do not rely on a fully automatic program specializer, but we delegate some work to the programmer to change our query interpreter into a program that specializes itself by treating queries as static data and data files as dynamic input. In particular, we use the following variant of the first Futamura projection:

```
target   = staged-interpreter(source)
```

Here, `staged-interpreter` is a version of the interpreter that has been *annotated* by the programmer. This idea was also used in bootstrapping the first implementation of the Futamura projections by Neil Jones and others in Copenhagen [8]. The role of the programmer can be understood as being part of the `mix` system, but we will see that the job of converting a straightforward interpreter into a staged interpreter is relatively easy.

Lightweight Modular Staging Staging or multi-stage programming describes the idea of making different computation stages explicit in a program, where the *present stage* program generates code to run in a *future stage*. The concept goes back at least to Jørring and Scherlis [9], who observed that many programs can be separated into stages, distinguished by frequency of execution or by availability of data. Taha and Sheard [22] introduced the language MetaML and made the case for making such stages explicit in the programming model through the use of quotation operators, as known from LISP and Scheme macros.

Lightweight modular staging (LMS) [17] is a staging technique based on types: instead of syntactic quotations, we use the Scala type system to designate future stage expressions. Where any regular Scala expression of type `Int`, `String`, or in general `T` is executed normally, we introduce a special type constructor `Rep[T]` with the property that all operations on `Rep[Int]`, `Rep[String]`, or `Rep[T]` objects will generate code to perform the operation later.

Here is a simple example of using LMS:

```
val driver = new LMS_Driver[Int,Int] {

  def power(b: Rep[Int], x: Int): Rep[Int] =
    if (x == 0) 1 else b * power(b, x - 1)

  def snippet(x: Rep[Int]): Rep[Int] = {
    power(x,4)
  }
}
driver(3)
↪ 81
```

We create a new `LMS_Driver` object. Inside its scope, we can use `Rep` types and corresponding operations. Method `snippet` is the 'main' method of this object. The driver will execute `snippet` with a symbolic input. This will completely evaluate the recursive power invocations (since it is a present-stage function) and record the individual expression in the IR as they are encountered. On exit of `snippet`, the driver will compile the generated source code and load it as executable into the running program. Here, the generated code corresponds to:

```
class Anon12 extends ((Int)=>(Int)) {
  def apply(x0:Int): Int = {
    val x1 = x0*x0
    val x2 = x0*x1
    val x3 = x0*x2
    x3
  }
}
```

The performed specializations are immediately clear from the types: in the definition of `power`, only the base `b` is dynamic (type `Rep[Int]`), everything else will be evaluated statically, at code generation time. The expression `driver(3)` will then execute the generate code, and return the result 81.

Some LMS Internals While not strictly needed to understand the rest of this paper, it is useful to familiarize oneself with some of the internals.

LMS is called *lightweight* because it is implemented as a library instead of baked-in into a language, and it is called *modular* because there is complete freedom to define the available operations on `Rep[T]` values. To user code, LMS provides just an abstract interface that lifts (selected) functionality of types `T` to `Rep[T]`:

```
trait Base {
  type Rep[T]
}
trait IntOps extends Base {
  implicit def unit(x: Int): Rep[Int]
  def infix_+(x: Rep[Int], y: Rep[Int]): Rep[Int]
  def infix_*(x: Rep[Int], y: Rep[Int]): Rep[Int]
}
```

Internally, this API is wired to create an intermediate representation (IR) which can be further transformed and finally unparsed to target code:

```
trait BaseExp {
  // IR base classes: Exp[T], Def[T]
  type Rep[T] = Exp[T]
  def reflectPure[T](x:Def[T]): Exp[T] = .. // insert x into IR graph
}
trait IntOpsExp extends BaseExp {
  case class Plus(x: Exp[Int], y: Exp[Int]) extends Def[Int]
  case class Times(x: Exp[Int], y: Exp[Int]) extends Def[Int]
  implicit def unit(x: Int): Rep[Int] = Const(x)
  def infix_+(x: Rep[Int], y: Rep[Int]) = reflectPure(Plus(x,y))
  def infix_*(x: Rep[Int], y: Rep[Int]) = reflectPure(Times(x,y))
}
```

Another way to look at this structure is as combining a shallow and a deep embedding for an IR object language [21]. Methods like `infix_+` can serve as smart constructors that perform optimizations on the fly while building the IR [18]. With some tweaks to the Scala compiler (or alternatively using Scala macros) we can extend this approach to lift language built-ins like conditionals or variable assignments into the IR, by redefining them as method calls [15].

Mixed-Stage Data Structures We have seen above that LMS can be used to unfold functions and generate specialized code based on static values. One key design pattern that will drive the specialization of our query engine is the notion of mixed-stage data structures, which have both static and dynamic components.

Looking again at our earlier `Record` abstraction:

```
case class Record(fields: Vector[String], schema: Vector[String]) {
  def apply(name: String): String = fields(schema indexOf name)
}
```

We would like to treat the schema as static data, and treat only the field values as dynamic. The field values are read from the input and vary per row, whereas the schema is fixed per file and per query. We thus go ahead and change the definition of `Record`s like this:

```
case class Record(fields: Vector[Rep[String]], schema: Vector[String]) {
  def apply(name: String): Rep[String] = fields(schema indexOf name)
}
```

Now the individual fields have type `Rep[String]` instead of `String` which means that all operations that touch any of the fields will need to become dynamic as well. On the other hand, all computations that only touch the schema will be computed at code generation time. Moreover, `Record` objects are static as well. This means that the generated code will manipulate the field values as individual local variables, instead of through a record indirection. This is a strong guarantee: records cannot exist in the generated code, unless we provide an API for `Rep[Record]` objects.

Staged Interpreter As it turns out, this simple change to the definition of records is the only significant one we need to make to obtain a query compiler from our previous interpreter. All other modifications follow by fixing the type errors that arise from this change. We show the full code again in Figure 4. Note that we are now using a staged version of the `Scanner` implementation, which needs to be provided as an LMS module.

```
val driver = new LMS_Driver[Unit,Unit] {
  type Fields = Vector[Rep[String]]
  type Schema = Vector[String]

  case class Record(fields: Fields, schema: Schema) {
    def apply(name: String):  Rep[String] = fields(schema indexOf name)
    def apply(names: Schema): Fields      = names map (this apply _)
  }

  def processCSV(file: String)(yld: Record => Unit): Unit = {
    val in = new Scanner(file)
    val schema = in.next('\n').split(",").toVector
    while (in.hasNext) {
      val fields = schema.map(n=>in.next(if(n==schema.last)'\n'else','))
      yld(Record(fields, schema))
    }
  }

  def evalRef(p: Ref)(rec: Record): Rep[String] = p match {
    case Value(a: String) => a
    case Field(name) => rec(name)
  }

  def evalPred(p: Predicate)(rec: Record): Rep[Boolean] = p match {
    case Eq(a,b) => evalRef(a)(rec) == evalRef(b)(rec)
    case Ne(a,b) => evalRef(a)(rec) != evalRef(b)(rec)
  }

  def execOp(o: Operator)(yld: Record => Unit): Unit = o match {
    case Scan(filename) =>
      processCSV(filename)(yld)
    case Print(parent) =>
      execOp(parent) { rec =>
        printFields(rec.fields) }
    case Filter(pred, parent) =>
      execOp(parent) { rec =>
        if (evalPred(pred)(rec)) yld(rec) }
    case Project(newSchema, parentSchema, parent) =>
      execOp(parent) { rec =>
        yld(Record(rec(parentSchema), newSchema)) }
    case Join(left, right) =>
      execOp(left) { rec1 =>
        execOp(right) { rec2 =>
          val keys = rec1.schema intersect rec2.schema
          if (rec1(keys) == rec2(keys))
            yld(Record(rec1.fields ++ rec2.fields, rec1.schema ++ rec2.schema)) }}
  }

  def printFields(fields: Fields) =
    printf(fields.map(_ => "%s").mkString("",",","\n"), fields: _*)

  def snippet(x: Rep[Unit]): Rep[Unit] = {
    val ops = parseSql("select room,title from talks.csv where time = '09:00 AM'")
    execOp(PrintCSV(ops)) { _ => }
  }
}
```

Figure 4. Staged query interpreter = compiler. Changes are underlined.

Results Let us compare the generated code to the one that was our starting point in Section 2. Our example query was:

```
select room, title from talks.csv where time = '09:00 AM'
```

And here is the handwritten code again:

```
printf("room,title")
val in = new Scanner("talks.csv")
in.next('\n')
while (in.hasNext) {
  val tid   = in.next(',')
  val time  = in.next(',')
  val title = in.next(',')
  val room  = in.next('\n')
  if (time == "09:00 AM")
    printf("%s,%s\n",room,title)
}
in.close
```

The generated code from the compiling engine is this:

```
val x1 = new scala.lms.tutorial.Scanner("talks.csv")
val x2 = x1.next('\n')
val x14 = while ({
  val x3 = x1.hasNext
  x3
}) {
  val x5 = x1.next(',')
  val x6 = x1.next(',')
  val x7 = x1.next(',')
  val x8 = x1.next('\n')
  val x9 = x6 == "09:00 AM"
  val x12 = if (x9) {
    val x10 = printf("%s,%s\n",x8,x7)
  } else {
  }
  x1.close
}
```

So, modulo syntactic differences, we have generated exactly the same code! And, of course, this code will run just as fast. Looking again at the Google Books query, where the interpreted engine tooks 45s to run the query, we are down again to 10s but this time *without giving up on generality*!

4. Beyond Simple Compilation

While we have seen impressive speedups just through compilation of queries, let us recall from Section 1 that we can still go faster. By writing our query by hand in C instead of Scala we were able to run it in 3s instead of 10s. The technique there was to use the `mmap` system call to map the input file into memory, so that we could treat it as a simple array instead of copying data from read buffers into string objects.

We have also not yet looked at efficient join algorithms that require auxiliary data structures, and in this section we will show how we can leverage generative techniques for this purpose as well.

Hash Joins We consider extending our query engine with hash joins and aggregates first. The required additions to `execOp` are straightforward:

```
def execOp(o: Operator)(yld: Record => Unit): Unit = o match {
  // ... pre-existing operators elided
  case Group(keys, agg, parent) =>
    val hm = new HashMapAgg(keys, agg)
    execOp(parent) { rec =>
      hm(rec(keys)) += rec(agg)
    }
    hm foreach { (k,a) =>
      yld(Record(k ++ a, keys ++ agg))
    }
  case HashJoin(left, right) =>
    val keys = resultSchema(left) intersect resultSchema(right)
    val hm = new HashMapBuffer(keys, resultSchema(left))
    execOp(left) { rec1 =>
      hm(rec1(keys)) += rec1.fields
    }
    execOp(right) { rec2 =>
      hm(rec2(keys)) foreach { rec1 =>
        yld(Record(rec1.fields ++ rec2.fields,
                   rec1.schema ++ rec2.schema))
      }
    }
}
```

An aggregation will collect all records from the parent operator into buckets, and accumulate sums in a hash table. Once all records are processed, all key-value pairs from the hash map will be emitted as records. A hash join will insert all records from the left parent into a hash map, indexed by the join key. Afterwards, all the records from the right will be used to lookup matching left records from the hash table, and the operator will pass combined records on to its callback. This approach is much more efficient for larger data sets than the naive nested loops join from Section 2.

Data Structure Specialization What are the implementations of hash tables that we are using here? We could have opted to just use lifted versions of the regular Scala hash tables, i.e. `Rep[HashMap[K,V]]` objects. However, these are not the most efficient for our case, since they have to support a very generic programming interface. Morever, recall our staged `Record` definition:

```
case class Record(fields: Vector[Rep[String]], schema: Vector[String]) {
  def apply(name: String): Rep[String] = fields(schema indexOf name)
}
```

A key design choice was to treat records as a purely staging-time abstraction. If we were to use `Rep[HashMap[K,V]]` objects, we would have to use `Rep[Record]` objects as well, or at least `Rep[Vector[String]]`. The choice of using `Vector[Rep[String]]` means that all field values will be mapped to individual entities in the generated code. This property naturally leads to a design for data structures in *column-oriented* instead of *row-oriented* order. Instead of working with:

```
Collection[ { Field1, Field2, Field3 } ]
```

We work with:

```
{ Collection[Field1], Collection[Field2], Collection[Field3] }
```

This layout has other important benefits, for example in terms of memory bandwidth utilization and is becoming increasingly popular in contemporary in-memory database systems.

Usually, programming in a columnar style is more cumbersome than in a record oriented manner. But fortunately, we can completely hide the column oriented nature of our internal data structures behind a high-level record oriented interface. Let us go ahead and implement a growable `ArrayBuffer`, which will serve as the basis for our `HashMaps`:

```
abstract class ColBuffer
case class IntColBuffer(data: Rep[Array[Int]]) extends ColBuffer
case class StringColBuffer(data: Rep[Array[String]],
                           len: Rep[Array[Int]]) extends ColBuffer

class ArrayBuffer(dataSize: Int, schema: Schema) {
  val buf = schema.map {
    case hd if isNumericCol(hd) =>
      IntColBuffer(NewArray[Int](dataSize))
    case _ =>
      StringColBuffer(NewArray[String](dataSize),
                      NewArray[Int](dataSize))
  }
  var len = 0
  def +=(x: Fields) = {
    this(len) = x
    len += 1
  }
  def update(i: Rep[Int], x: Fields) = (buf,x).zipped.foreach {
    case (IntColBuffer(b), RInt(x)) => b(i) = x
    case (StringColBuffer(b,l), RString(x,y)) => b(i) = x; l(i) = y
  }
  def apply(i: Rep[Int]): Fields = buf.map {
    case IntColBuffer(b) => RInt(b(i))
    case StringColBuffer(b,l) => RString(b(i),l(i))
  }
}
```

The array buffer is passed a schema on creation, and it sets up one `ColBuffer` object for each of the columns. In this version of our query engine we also introduce typed columns, treating columns whose name starts with "#" as numeric. This enables us to use primitive integer arrays for storage of numeric columns instead of a generic binary format. It would be very easy to introduce further specialization, for example sparse or compressed columns for

cases where we know that most values will be zero. The `update` and `apply` methods of `ArrayBuffer` still provide a row-oriented interface, working on a set of `Fields` together, but internally access the distinct column buffers.

With this definition of array buffers at hand, we can define a class hierarchy of hash maps, with a common base class and then derived classes for aggregations (storing scalar values) and joins (storing collections of objects):

```
class HashMapBase(keySchema: Schema, schema: Schema) {
  val keys = new ArrayBuffer(keysSize, keySchema)
  val htable = NewArray[Int](hashSize)
  def lookup(k: Fields) =
  def lookupOrUpdate(k: Fields)(init: Rep[Int]=>Rep[Unit]) = ...
}
// hash table for groupBy, storing scalar sums
class HashMapAgg(keySchema: Schema, schema: Schema) extends
      HashMapBase(keySchema: Schema, schema: Schema) {
  val values = new ArrayBuffer(keysSize, schema)

  def apply(k: Fields) = new {
    def +=(v: Fields) = {
      val keyPos = lookupOrUpdate(k) { keyPos =>
        values(keyPos) = schema.map(_ => RInt(0))
      }
      values(keyPos) = (values(keyPos) zip v) map {
        case (RInt(x), RInt(y)) => RInt(x + y)
      }}}
  def foreach(f: (Fields,Fields) => Rep[Unit]): Rep[Unit] =
    for (i <- 0 until keyCount)
      f(keys(i),values(i))
}
// hash table for joins, storing lists of records
class HashMapBuffer(keySchema: Schema, schema: Schema) extends
      HashMapBase(keySchema: Schema, schema: Schema) {
        // ... details elided
}
```

Note that the hash table implementation is oblivious of the storage format used by the array buffers. Furthermore, we're freely using object oriented techniques like inheritance without the usually associated overheads because all these abstractions exist only at code generation time.

Memory-Mapped IO and Data Representations Finally, we consider our handling of memory mapped IO. One key benefit will be to eliminate data copies and represent strings just as pointers into the memory mapped file, instead of first copying data into another buffer. But there is a problem: the standard C API assumes that strings are 0-terminated, but in our memory mapped file, strings will be delimited by commas or line breaks. To this end, we introduce our own operations and data types for data fields. Instead of the previous definition of `Fields` as `Vector[Rep[String]]`, we introduce a small class hierarchy `RField` with the necessary operations:

```
type Fields = Vector[RField]
abstract class RField {
  def print()
  def compare(o: RField): Rep[Boolean]
  def hash: Rep[Long]
}
case class RString(data: Rep[String], len: Rep[Int]) extends RField {
  def print() = ...
  def compare(o: RField) = ...
  def hash = ...
}
case class RInt(value: Rep[Int]) extends RField {
  def print() = printf("%d",value)
  def compare(o: RField) = o match { case RInt(v2) => value == v2 }
  def hash = value.asInstanceOf[Rep[Long]]
}
```

Note that this change is again completely orthogonal to the actual query interpreter logic.

As the final piece in the puzzle, we provide our own specialized `Scanner` class that generates `mmap` calls (supported by a corresponding LMS IR node), and creates `RField` instances when reading the data:

```
class Scanner(name: Rep[String]) {
  val fd = open(name)
  val fl = filelen(fd)
  val data = mmap[Char](fd,fl)
  var pos = 0
  def next(d: Rep[Char]) = {
    //...
    RString(stringFromCharArray(data,start,len), len)
  }
  def nextInt(d: Rep[Char]) = {
    //...
    RInt(num)
  }
}
```

With this, we are able to generate tight C code that executes the Google Books query in 3s, just like the hand written optimized C code. The total size of the code is just under 500 (non-blank, non-comment) lines.

The crucial point here is that while we cannot hope to beat hand-written *specialized* C code for a particular query–after all, anything we generate could also be written by hand–we are beating, by a large margin, the highly optimized *generic* C code that makes up the bulk of MySQL and other traditional database systems. By changing the perspective to embrace a generative approach we are able to raise the level of abstraction, and to leverage high-level functional programming techniques to achieve excellent performance with very concise code.

5. Perspectives

This paper is a case study in "abstraction without regret": achieving high performance from very high level code. More generally, we argue for a radical rethinking of the role of high-level languages in performance critical code [16]. While our work demonstrates that Scala is a good choice, other expressive modern languages can be used just as well, as demonstrated by Racket macros [23], DSLs Accelerate [12], Feldspar [1], Nikola [11] (Haskell), Copperhead [2] (Python), Terra [4, 5] (Lua).

Our case study illustrates a few common generative design patterns: higher-order functions for composition of code fragments, objects and classes for mixed-staged data structures and for modularity at code generation time. While these patterns have emerged and proven useful in several projects, the field of practical generative programming is still in its infancy and is lacking an established canon of programming techniques. Thus, our plea to language designers and to the wider PL community is to ask, for each language feature or programming model: "how can it be used to good effect in a generative style?"

References

[1] E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. The design and implementation of feldspar: An embedded language for digital signal processing. IFL'10, 2011.

[2] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. PPoPP, 2011.

[3] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *POPL*, 1993.

[4] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: a multi-stage language for high-performance computing. In *PLDI*, 2013.

[5] Z. DeVito, D. Ritchie, M. Fisher, A. Aiken, and P. Hanrahan. First-class runtime generation of high-performance types using exotypes. In *PLDI*, 2014.

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

[7] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[8] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[9] U. Jørring and W. L. Scherlis. Compilers and staging transformations. In *POPL*, 1986.

[10] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[11] G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in Haskell. Haskell, 2010.

[12] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional GPU programs. ICFP, 2013.

[13] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[14] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[15] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-virtualized: Linguistic reuse for deep embeddings. Higher-Order and Symbolic Computation (Special issue for PEPM'12).

[16] T. Rompf, K. J. Brown, H. Lee, A. K. Sujeeth, M. Jonnalagedda, N. Amin, G. Ofenbeck, A. Stojanov, Y. Klonatos, M. Dashti, C. Koch, M. Püschel, and K. Olukotun. Go meta! A case for generative programming and dsls in performance critical systems. In *SNAPL*, 2015.

[17] 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.

[18] T. Rompf, A. K. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs. POPL, 2013.

[19] M. Stonebraker and U. Çetintemel. "One Size Fits All": An idea whose time has come and gone (abstract). In *ICDE*, pages 2–11, 2005.

[20] 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.

[21] J. Svenningsson and E. Axelsson. Combining deep and shallow embedding for EDSL. In *TFP*, 2012.

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

[23] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. PLDI, 2011.

[24] 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.