Development of a fast Domain-Specific Language: A DSL for Stream Processing

Master Semester Project Report Fall 2012

Vera Salvisberg, vera@salvisberg.name

Supervisor: Tiark Rompf, LAMP, EPFL

1 Introduction

Domain-specific languages (DSLs) are an active area of research, because they enable domain experts to express their problems in a simple and concise manner, without having to be programming experts as well. While the advantage of having a language tailored to fit a domain is obvious as far as the productivity of the programmer using the DSL is concerned, we also need to consider the performance of the resulting system, and the cost of developing the DSL.

There are two broad categories of DSLs, stand-alone and embedded. For a stand-alone DSL, the whole toolchain needs to be implemented: the grammar, the parser, the compiler-linker or an interpreter with a virtual machine. An embedded DSL on the other side is programmed as a library in an existing host language, and takes advantage of the existing infrastructure. While the additional layers of indirection have a performance penalty, most applications don't justify the effort of developing a stand-alone DSL.

The LMS [1] and Delite [2] frameworks developed at EPFL provide the best from both worlds: while the DSL can be programmed in Scala, the frameworks allow for code generation that removes the layers of indirection, and provide constructs for parallelizing code with minimal effort. Developing a DSL becomes easy through the existing libraries and semantic building blocks.

Many problems profit considerably from parallelization and can be expressed elegantly in a DSL that does parallelization without the programmer having to worry about it (think map-reduce). In my semester project, I explore a different class of problems, that are more naturally expressed through a stream processing framework. There are one or several components that input data; blocks that transform, aggregate or distribute data; and components that consume the processed data.

Some concrete examples are:
- signal processing: applying a FIR filter to sensor data to compute an aggregate
- statistics: displaying the sales within the last hour of an e-commerce
- math: programming a complex function and displaying the output as a function of the input
- databases: updating query results on changes without recomputation (DBToaster [3])

2 Structure

All code is here: https://github.com/vsalvis/DslStreams/tree/report/ScalaStreams.

I started by implementing some common stream components in *streams/Streams.scala*. The abstract class defining a stream is as follows:

```
abstract class StreamOp[A] {
  def onData(data: A)
  def flush
}
```

A `StreamOp` is a black box processing data of the generic type `A`. On every new input, `onData` is called. It might for example print the data, or transform it and send it to a next `StreamOp`. The `flush` method is used to pass a reset signal through the streams. The exact semantics depend on the `StreamOp`.

To input data into my streams, I added the `StreamInput` abstract class, and as a concrete example the `ListInput` class which feeds all elements of a `List` to the next `StreamOp`:

```
abstract class StreamInput[A](stream: StreamOp[A])

class ListInput[A](input: List[A], stream: StreamOp[A]) extends StreamInput[A](stream) {
  input foreach stream.onData

  def flush = {
    stream.flush
  }
}
```

Similarly, there's a `StreamOutput` abstract class for displaying the data processed by the stream, and an example that prints each element on a new line:

```
abstract class StreamOutput[A] extends StreamOp[A] {
  def flush = {}
}

class PrintlnOp[A] extends StreamOutput[A] {
  def onData(data: A) = println(data)
}
```

Now that the start and end of the stream processing pipeline are in place, let's look at some examples of stream operations.

```
class MapOp[A, B](f: A => B, next: StreamOp[B]) extends StreamOp[A] {
  def onData(data: A) = next.onData(f(data))

  def flush = next.flush
}
```

```
class FilterOp[A](p: A => Boolean, next: StreamOp[A]) extends StreamOp[A] {
  def onData(data: A) = if (p(data)) next.onData(data)

  def flush = next.flush
}

class DuplicateOp[A](next1: StreamOp[A], next2: StreamOp[A]) extends StreamOp[A] {
  def onData(data: A) = {
    next1.onData(data)
    next2.onData(data)
  }

  def flush = {
    next1.flush
    next2.flush
  }
}
```

Example 1: With these three operations we can already solve a simple problem: multiply all input elements by 3, separate the even from the odd ones and multiply the even ones by 2 and the odd ones by 3:

```
new ListInput(List.range(0, 6),
    new MapOp({x: Int => 3 * x},
        new DuplicateOp(
            new FilterOp({x: Int => x % 2 == 0},
                new MapOp({x: Int => 2 * x + " (even)"}, new PrintlnOp)),
            new FilterOp({x: Int => x % 2 == 1},
                new MapOp({x: Int => 3 * x + " (odd)"}, new PrintlnOp)))))
```

With an input of List(0, 1, 2, 3, 4, 5) we get the following output:

```
0 (even)
9 (odd)
12 (even)
27 (odd)
24 (even)
45 (odd)
```

While the example does what it's supposed to, the code isn't very readable. The next section presents the API that enables a more intuitive formulation of stream composition.

3 API

The API in *streams/StreamApi.scala* introduces a new `Stream` class, which has nicer syntax, but also different semantics which allow streams to be composed "from left to right". When instantiating a `StreamOp`, all following streams already have to be known, since they have to be provided as constructor arguments. As a consequence, the `StreamOp`s are composed from right to left, and while you can always add a new `StreamOp` at the beginning of the stream, you cannot modify the end:

```
val stream = new MapOp({x: Int => 2 * x}, new PrintlnOp)
val stream2 = new FilterOp({x: Int => x % 2 == 0}, stream) // Adding on the left
// cannot add on right, stream finishes with PrintlnOp
```

Here's the `Stream` class with the operations we know already:

```
abstract class Stream[A,B] { self =>
  def into(out: StreamOp[B]): StreamOp[A]

  def filter(p: B => Boolean) = new Stream[A,B] {
    def into(out: StreamOp[B]) = self.into(new FilterOp(p, out))
  }
  def map[C](f: B => C) = new Stream[A,C] {
    def into(out: StreamOp[C]) = self.into(new MapOp(f, out))
  }
  def duplicate(first: StreamOp[B], second: StreamOp[B]) = {
    self.into(new DuplicateOp(first, second))
  }

  def print = into(new PrintListOp())
}

object Stream {
  def apply[A] = new Stream[A,A] {
    def into(out: StreamOp[A]): StreamOp[A] = out
  }
}
```

It adds an intermediary step of stream construction before actually instantiating the needed `StreamOp`s (with explicit types for clarity):

```
val streamAPI1: Stream[Int, Int] = Stream[Int] map {2 * _}
val streamAPI2: Stream[Int, Int] = streamAPI1 filter {_ % 2 == 0} // Adding on the right
// cannot add on left before finishing the stream
val streamAPI3: StreamOp[Int] = streamAPI2 print // Instantiating the StreamOps
```

Example 1b: The lighter syntax also lets us express the previous example much more concisely:

```
new ListInput(List.range(0, 6),
    Stream[Int] map {3 * _} duplicate (
        Stream[Int] filter {_ % 2 == 0} map {2 * _ + " (even)"} print,
        Stream[Int] filter {_ % 2 == 1} map {3 * _ + " (odd)"} print))
```

The complete API contains functions known from the collections API, and operations like `groupBy` and `equiJoin` from the SQL world:

```
abstract class Stream[A,B] { self =>
  def into(out: StreamOp[B]): StreamOp[A]

  def print = into(new PrintlnOp())

  def aggregate() = new Stream[A,List[B]] {…}
  def drop(n: Int) = new Stream[A,B] {…}
  def dropWhile(p: B => Boolean) = new Stream[A,B] {…}
  def filter(p: B => Boolean) = new Stream[A,B] {…}
  def flatMap[C](f: B => List[C]) = new Stream[A,C] {…}
  def fold[C](f: (B, C) => C, z: C) = new Stream[A,C] {…}
  def groupByStream[K](keyF: B => K) = new Stream[A,Map[K, List[B]]] {…}
  def map[C](f: B => C) = new Stream[A,C] {…}
  def multiSplit[C](num: Int, streams: (StreamOp[C], Int) => StreamOp[B])
      = new Stream[A, List[C]] {…}
  def offset(n: Int) = new Stream[A,B] {…}
  def prepend(list: List[B]) = new Stream[A,B] {…}
  def reduce(f: (B, B) => B) = new Stream[A,B] {…}
  def splitMerge[C,D](first: Stream[B,C], second: Stream[B,D])
      = new Stream[A,Pair[C,D]] {…}
  def take(n: Int) = new Stream[A,B] {…}
  def takeWhile(p: B => Boolean) = new Stream[A,B] {…}

  // special functions: Those are not simple Streams because they have
  // multiple in- or output streams
  def duplicate(first: StreamOp[B], second: StreamOp[B]) = {…}
  def equiJoin[C, D, K](other: Stream[C,D], keyFunThis: B => K,
      keyFunOther: D => K, next: StreamOp[List[(B,D)]]) = {…}
  def groupBy[K](keyF: B => K, streamF: K => StreamOp[B]) = {…}
  def multiZipWith(num: Int, others: List[Stream[A,B]],
      next: StreamOp[List[B]]) = {…}
  def zipWith[C,D](other: Stream[C,D], next: StreamOp[Pair[B, D]]) = {…}
}
```

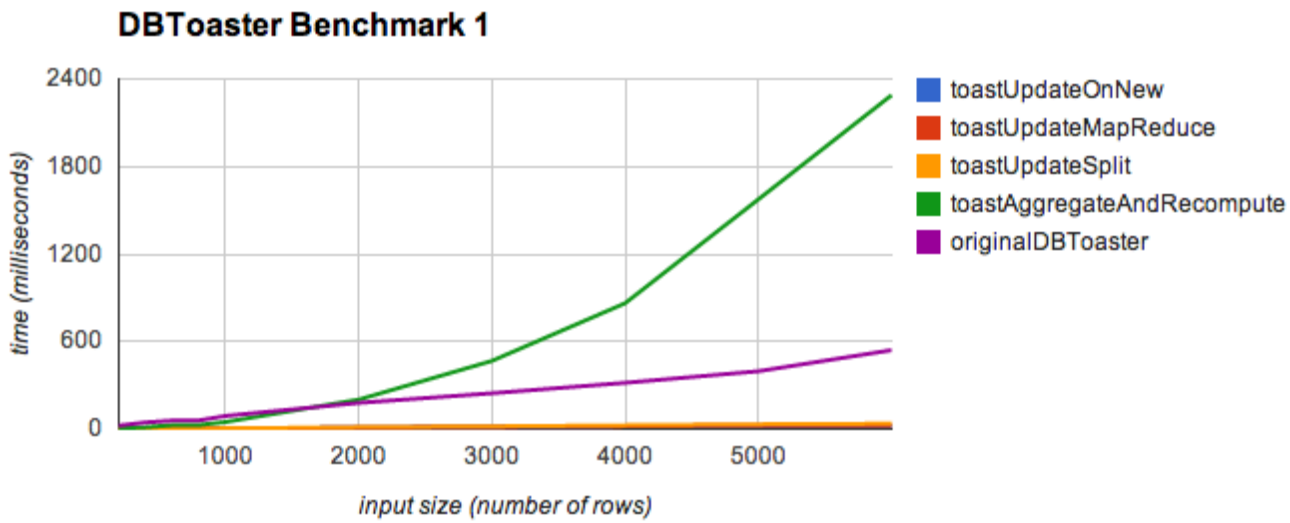
4 Applications and Benchmarks

In the *streams/Applications.scala* file I implemented several versions of a FIR filter as well as low-pass and band-pass filters to show how the stream library can be used to do signal processing.

But it can just as well be used for doing delta-processing on database queries, as is demonstrated in *streams/StreamDBToaster.scala*. Delta-processing is based on the observation that an up-to-date query result often doesn't need to be recomputed from the complete data set on every modification, but can be computed from the last result and the modification. I implemented the following simple SQL-query (TPCH query 1):

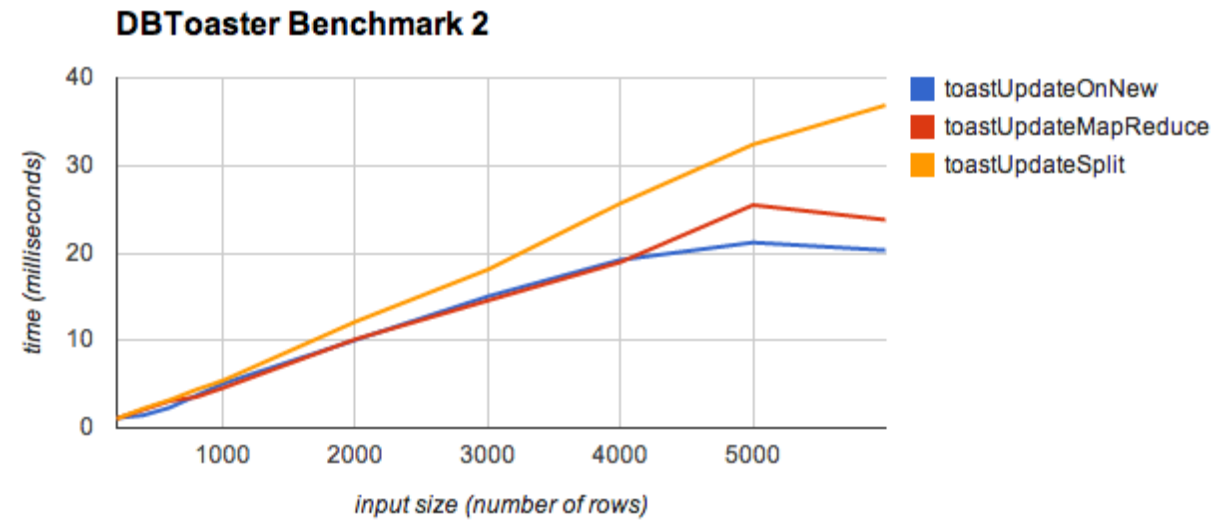
```
SELECT returnflag, linestatus,
  SUM(quantity) AS sum_qty,
  SUM(extendedprice) AS sum_base_price,
  SUM(extendedprice * (1-discount)) AS sum_disc_price,
  SUM(extendedprice * (1-discount)*(1+tax)) AS sum_charge,
  AVG(quantity) AS avg_qty,
  AVG(extendedprice) AS avg_price,
  AVG(discount) AS avg_disc,
  COUNT(*) AS count_order
FROM lineitem
WHERE shipdate <= DATE('1997-09-01')
GROUP BY returnflag, linestatus;
```

The DBToaster [3] project at EPFL explores this idea in general, and I compared my streams (blue, red and yellow) to the scala code generated by DBToaster (*dbtoaster/DBToasterQuery1.scala*, violet). As a base line, I also implemented the naive solution which aggregates all the data and recomputes the output from scratch on every change (green). In the first benchmark chart below you can see that the DBToaster code runs slower than the naive solution up to about 1500 input rows.



I expect a linear growth for my streams and DBToaster, but an exponential one for the naive solution. The benchmark seems to confirm that expectation.

The second benchmark chart shows my three different stream implementations, and we see that my streams are **an order of magnitude faster** than DBToaster. This is a great result, but it also isn't very surprising, since DBToaster is designed to solve a much more general problem, and the machine-generated DBToaster code is 4000 lines long vs. 200 lines of specialized hand-written code for my four implementations.



The three implementations have comparable runtime. All start with a `FilterOp` on the date to do the SQL WHERE operation and then a `GroupByOp` for the GROUP BY. Each group is then sent to a new instance of the stream chain computing the averages, sums and the count. `ToastUpdateOnNew` just uses a specialized `StreamOp` that stores the last result and updates it on every new entry. Slightly slower is `toastUpdateMapReduce`, which uses the existing `MapOp` and `ReduceOp`. The slowest one is `toastUpdateSplit`, which splits the stream into 8 streams that then each do just an average or sum on their number. Conceptually, this is the cleanest solution, but it has a performance penalty of

roughly a factor 2. The next section explains how we can get rid of some of that penalty.

6 Performance

So far, my stream DSL is just a regular Scala library. But as we have seen in the last section, there is much room for performance improvement. The LMS [1] framework developed at EPFL can be used to generate code and apply domain specific optimizations. While the DSL programmer can still use the high-level abstractions, several stages of compilation transform the program to a faster version.

As an example, I implemented some optimizations on `MapOps` in *dsl/DSL.scala*. With this, multiple successive `MapOps` are unfolded into one. The following `test` function creates a stream that first squares its input and then adds 3 to it:

```
def test(s: Rep[DoubleStream]): Rep[DoubleStream] =
  map(map(s, {(x: Rep[Double]) => Math.pow(unit(2.0), x)}),
      {(x: Rep[Double]) => x + unit(3.0)})
```

From it, the following code is generated without using the unfolding optimization:

```
// Code generated by Test without optimization:
class Test extends (Stream[Double, Double] => Stream[Double, Double]) {
  def apply(x9: Stream[Double, Double]): Stream[Double, Double] = {
    val x12 = {x10: (Double) =>
      val x11 = java.lang.Math.pow(2.0, x10)
      x11: Double
    }
    val x13 = x9.map(x12)        // first MapOp
    val x16 = {x14: (Double) =>
      val x15 = x14 + 3.0
      x15: Double
    }
    val x17 = x13.map(x16)       // second MapOp
    x17
  }
}
```

With the optimization, only one `MapOp` is used, with the combined function:

```
// Code generated by Test with optimization:
class TestOpt extends (Stream[Double, Double] => Stream[Double, Double]) {
  def apply(x8: Stream[Double, Double]): Stream[Double, Double] = {
    val x11 = {x9: (Double) =>
      val x10 = java.lang.Math.pow(2.0,x9)
      x10: Double
    }
    val x16 = {x13: (Double) =>
      val x14 = x11(x13)
      val x15 = x14 + 3.0
      x15: Double
    }
    val x17 = x8.map(x16)       // only one MapOp
    x17
  }
}
```

We can also directly use the generated test function in regular Scala code:

```
new streams.ListInput(1.0 :: 2.0 :: 3.0 :: Nil, test(Stream[Double]) print)
```

And we get the desired output:

```
5.0
7.0
11.0
```

7 Conclusion and Future Work

While this project helped me in understanding the decisions involved with designing an API, I only had time to dive very shallowly into LMS. In the follow-up project I want to explore more performance optimizations possible with LMS. Some ideas include unfolding of map, reduce/fold and filter operations, replacement of split-merge operations and more generally removal of buffers in zip operations.

As a second idea, I would like to explore how Delite [2] could be used to generate parallel code. Eventually, it would be nice to have Stream operations publicly available as building blocks in LMS/Delite.

In conclusion, I liked the project a lot because it was very diverse and let me build my own DSL, and I even had time to take some first steps in LMS. Thanks to everybody at LAMP!

8 References

[1] T. Rompf, M. Odersky: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs (GPCE 2010)
[2] H. Lee, K. Brown, A. Sujeeth, H. Chafi, T. Rompf, M. Odersky, K. Olukotun: Implementing Domain-Specific Languages for Heterogeneous Parallel Computing (IEEE 2011)
[3] Y. Ahmad, O. Kennedy, C. Koch, M. Nikolic: DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views (VLDB 2012)