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

Part II: Lifting the DSL

Master Optional Semester Project Report Spring 2013

Vera Salvisberg, vera@salvisberg.name

Supervisor: Tiark Rompf, LAMP, EPFL

1 Introduction

In my first semester project[1], I developed a Scala-embedded domain-specific language (DSL) for Stream Processing. The stream paradigm is a computational model that is fundamentally different from the well-researched and understood data parallelism paradigm. Programming models like map-reduce act on datasets that are completely available at runtime, and can therefore parallelize work by splitting the dataset into parts and process each part separately, before combining the results. In the stream processing world, the data comes in continuously (for example from an external sensor, a server or even the file system), and the processing framework transforms it by running it through a combination of boxes with well-defined simple functionality.

A stream processing framework usually has several parts. There are one or several components that input data; blocks that transform, aggregate or distribute data; and components that consume the processed data, for example by displaying the result in a GUI.

Some concrete use cases 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])

The report is structured as follows:

[1] https://github.com/vsalvis/DslStreams/blob/master/ScalaStreams/doc/2012_Fall_DSL%20report.pdf

2 Previous work

a. Scala Streams

By the end of the first semester project[2], I had developed a Stream Processing DSL that was embedded in Scala (DSL as a library approach) and can be found in *streams/Streams.scala*. The abstract class defining a scala 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`. Some examples of simple `StreamOps` implemented in the project are:

```
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))))
```

[2] https://github.com/vsalvis/DslStreams/tree/report/ScalaStreams

b. Scala Streams API

Next, I implemented an API in *streams/StreamApi.scala*, with nicer syntax and better Stream composition abilities. Here's the `Stream` class with the subset of 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
  }
}
```

Example 1b: The lighter syntax 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))
```

c. Lifted Streams

The third step was to explore optimizations on this DSL. 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, multiple stages of compilation transform the program to a faster version. Last semester I did a proof of concept implementation in *dsl/DSL.scala*. With this, multiple successive `MapOps` and `ScaleOps` were unfolded into one `MapOp`.

In this new DSL, I was lifting the two `StreamOps` to get AST nodes for the `ScaleOp` and the `MapOp`, so I could then pattern match on combinations of those and fuse them. LMS distinguishes the stages of compilation through the type system, where Rep[T] indicates that code will be generated for T.

```
// Concepts and concrete syntax
trait StreamDSL { this: Base =>
  def scale[A](s: Rep[Stream[A,Double]], k: Rep[Double]): Rep[Stream[A,Double]]
  def map[A, B, C](s: Rep[Stream[A,B]], f: Rep[B] => Rep[C]): Rep[Stream[A,C]]
}

// Intermediate representation
trait StreamDSLExp extends StreamDSL with [...] { this: BaseExp =>
  case class Scale[A](s: Exp[Stream[A,Double]], k: Exp[Double])
      extends Def[Stream[A,Double]]
  case class Map[A,B,C](s: Exp[Stream[A,B]], f: Exp[B => C])
      extends Def[Stream[A,C]]
  [...]
}

// Optimizations working on the intermediate representation
trait StreamDSLOpt extends StreamDSLExp { this: BaseExp =>
  override def scale[A](s: Exp[Stream[A,Double]], k: Exp[Double]) = s match {
    case Def(Scale(s1, k1)) =>
      super.scale(s1, numeric_times(k, k1))
    case Def(Map(s1, Def(Lambda(f1,_,_)))) =>
      super.map(s1, {x => numeric_times(k, f1(x))})
    case _ => super.scale(s, k)
  }
  override def map[A, B, C](s: Exp[Stream[A,B]], f: Exp[B] => Exp[C]) = [...]
}
```

So the optimizations were done by hand, by pattern matching on the AST and fusing successive operations into one. While this approach works, it won't scale well, because of the combinatorial explosion of cases. Even with just the 20 `StreamOps` in the API, I'd need to treat 400 cases. Fortunately, this is exactly what LMS was designed to do, and the next section explains how I used it to generate optimized code automatically in this semester project.

3 StreamOps on Rep Types

a. Stateless RepStreams

This semester, I used LMS to do the code fusion. So instead of lifting the `StreamOps` by using a `Rep[Stream]`, I now created a DSL of Streams on Rep types.

```
trait RepStreamOps extends [...] {
  abstract class RepStreamOp[A] {
    def onData(data: Rep[A])
    def flush
  }
  class RepMapOp[A, B](f: Rep[A] => Rep[B], next: RepStreamOp[B])
      extends RepStreamOp[A] {
    def onData(data: Rep[A]) = next.onData(f(data))
    def flush = next.flush
  }
  [...]
}
```

The effect of this is that the code generated will not contain any streams, it will just be a single function containing the bodies of all the onData functions being called. For example, consider the following code in the `RepStream` DSL:

```
def onData1(i: Rep[Int]) = {
  val s = RepStream[Int].map({x: Rep[Int] => x * unit(2)})
                        .filter({x: Rep[Int] => x > unit(3)})
                        .flatMap({x: Rep[Int] => x :: (x + unit(1)) :: Nil})
                        .print
  s.onData(i)
}
```

This will be compiled to a single function. Note that the function is purely functional, in the sense that it doesn't have any state:

```
class onData1 extends ((Int)=>(Unit)) {
  def apply(x9:Int): Unit = {
    val x10 = x9 * 2
    val x11 = x10 > 3
    val x16 = if (x11) {
    val x13 = println(x10)
    val x12 = x10 + 1
    val x14 = println(x12)
    () } else { () } ()
  }
```

It can then be called from regular Scala code:

```
val scala_onData1 = concreteProg.compile(onData1)          4
                                                          5
scala_onData1(1)                                   =>     6
scala_onData1(2)                                          7
scala_onData1(3)                                          8
scala_onData1(4)                                          9
```

b. Stateful RepStreams

We need to distinguish between stateless and stateful stream ops. While the above example does what we'd expect, there's some more magic required for operations that have an internal state, like fold, reduce, take, drop etc. Let's consider the naive fold:

```
class RepFoldOp[A, B](f: (Rep[A], Rep[B]) => Rep[B], z: B, next: RepStreamOp[B])
    extends RepStreamOp[A] {
  var result = z
  def onData(data: Rep[A]) = { result = f(data, result); next.onData(result) }
  def flush = { result = z; next.flush }
}
```

This simple example program, when called with (1, 2, 3, 4), will output (2, 3, 4, 5) instead of (2, 4, 7, 11):

```
def onData2(i: Rep[Int]) = {
  val s = RepStream[Int].fold[Int]({(x, y) => x + y}, 1).print
  s.onData(i)
}
```

This happens because it is compiled to the following function, which doesn't keep the result between invocations:

```
class onData2 extends ((Int)=>(Unit)) {
  def apply(x18:Int): Unit = {
    val x19 = x18 + 1
    val x20 = println(x19)
    ()
  }
}
```

Luckily, LMS allows the use of static data, and with the following line we can tell it that we'll use an `Array` to store out changing state:

```
override def array_apply[T](x: Exp[Array[T]], n: Exp[Int]): Exp[T] =
  reflectWrite(x)(ArrayApply(x, n))
```

We can then express our fold as follows:

```
class RepFoldOp[A, B](f: (Rep[A], Rep[B]) => Rep[B], z: B, next: RepStreamOp[B])
    extends RepStreamOp[A] {
  val state = new Array[B](1)
  state(0) = z

  def onData(data: Rep[A]) = {
    val stateR: Rep[Array[B]] = staticData(state)
    val result = f(data, stateR(unit(0)))
    stateR(unit(0)) = result
    next.onData(result)
  }

  def flush = {
    val stateR: Rep[Array[B]] = staticData(state)
    stateR(unit(0)) = unit(z)
    next.flush
  }
}
```

Now, `onData2` is compiled to the following code, which will produce the correct values:

```
class onData2(px19:Array[Int]) extends ((Int)=>(Unit)) {
  def apply(x18:Int): Unit = {
    val x19 = px19 // static data: Array(1)
    val x20 = x19(0)
    val x21 = x18 + x20
    val x22 = x19(0) = x21
    val x23 = println(x21)
    ()
  }
}
```

LMS handles the allocation of the Array and wrapping of the call to pass it as argument, so the user code for this function is identical to the one in section a.

This DSL is already quite powerful, because the user programmer can write programs in terms of the high-level abstractions and compose the streams from the building blocks. LMS then takes care of stripping out all the boilerplate, like object allocations and virtual method calls. And it comes with non-domain specific optimizations like constant folding and loop fusion for free.

c. GroupBy Problem

However, not all stateful `RepStreamOps` are easy to persist. For `RepGroupByOp`, `SplitMergeOp`, `MultiSplitOp` and `zipWith`, the state we want to save are `RepStreamOps`. But we cannot store them in the LMS world, because there's no `Rep[RepStreamOp]`. We would have to stage all the Ops to be able to persist that state correctly.

In the case of the groupBy, we might not always want to create a HashMap, because it can't grow indefinitely, and lots of objects would be created since each input would cause an immutable copy of the current HashMap to be pushed into the next stream. So we would like to be able to push the elements individually into the stream corresponding to their key. As a compromise, I'll investigate having a `RepGroupByOp` with a fixed-size `Array[RepStreamOp]` as argument. So this basically means taking the hash function out of the hash map and into the key function, so that the user just provides a fixed number of groups.

Alternatively, to go with the LMS spirit, the receiver streams could be compiled during runtime, whenever a new key/group is encountered. We will investigate this idea for our upcoming paper submission.

4 WindowJoin

This semester, I also read multiple papers about window joins, e.g. a join that only looks at the last x elements of each input stream. Window Join is an interesting problem because it can't easily be parallelized by distributing the data over several cores, since all elements of one stream which are in the current window need to be joined with all current elements of the other stream, so distributing data over cores will result in lots of memory activity.

Teubner and Mueller describe a different approach in their paper "How Soccer Players Would do Stream Joins" [2]. The idea is to place the cores in a line, and let the data streams flow against each other, so that the elements flow from core to core, and are guaranteed to see all elements of the other stream if they're part of the same window. The paper sketches the idea as follows:

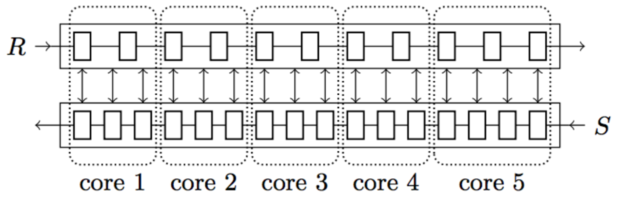

Figure 5: Parallelized handshake join evaluation. Each compute core processes one segment of both windows and performs all comparisons locally.

I implemented the algorithm in *windowJoin/WindowJoin.scala*. In future work, I would like to investigate how using streams between cores would influence performance. There's also a neat optimization of the algorithm I'd like to try out: instead of having all cores dump their results into a common queue, I'd like to pass the results intermixed in one of the streams, avoiding the concurrency bottleneck.

While this algorithm can't really be pushed completely into the LMS world because it's only really interesting when multiple cores can be used, it could be a nice example of where streams provide a speed-up.

5 Conclusion

In this second semester project, I was able to dive much deeper into the LMS world and really understand the slogan “abstraction without regrets”. While the semester is always over too quickly, I’ll keep working on this project for a paper submission. I’d again like to thank everybody at LAMP, especially Tiark for providing the big picture and helping me understand and debug some LMS problems, and Manohar and Vlad for their help at various points with Manifest magic and other points I got stuck on.

6 References

[1] T. Rompf, M. Odersky: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs (GPCE 2010)
[2] J. Teubner, R. Mueller: How Soccer Players Would do Stream Joins (SIGMOD 2011)