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

Part II: Lifting the DSL

Spring 2013 Semester Project Presentation

Student: Vera Salvisberg, Master IN, EPFL
Supervisor: Tiark Rompf, LAMP, EPFL
Date: June 13, 2013

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Outline

- Introduction
- Previous work
  - Scala Streams, API and manual optimizations
- StreamOps on Rep Types
  - Stateless RepStreams
  - Stateful RepStreams
  - GroupBy Problem
- WindowJoin
- Future Work
- Conclusion

Scala Streams

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

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
}
```

Example Usage, API

```
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)))))
```

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

API code:

```
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))
```

```
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)})
```

```
// Without optimization:
class Test extends (SDD=>SDD) {
  def apply(x0:SDD): SDD = {
    val x3 = {x1: (Double) =>
      val x2 = Math.pow(2.0,x1)
      x2: Double
    }
    val x4 = x0.map(x3)  // MapOp
      val x7 = {x5: (Double) =>
      val x6 = x5 + 3.0
      x6: Double
    }
    val x8 = x4.map(x7)  // MapOp
    x8
  }
}
```

```
// With optimization:
class TestOpt extends (SDD=>SDD) {
  def apply(x0:SDD): SDD = {
    val x8 = {x5: (Double) =>
      val x6 = java.lang.Math.pow(2.0,x5)
      val x7 = x6 + 3.0
      x7: Double
    }
    val x11 = x0.map(x18) // MapOp
    x11
  }
}
```

where SDD = Stream[Double, Double]

RepStreamOps

```scala
abstract class RepStreamOp[A] {
  def onData(data: Rep[A]): Unit
  def flush: Unit
}

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
}
```

RepStreamOps in action

```
def onData1(i: Rep[Int]) = {
  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.onData(i)
}
```

⇩

```
class onData1 extends ((Int)=>(Unit)) {
  def apply(x0:Int): Unit = {
    val x1 = x0 * 2
    val x2 = x1 > 3
    val x7 = if (x2) {
      val x4 = println(x1)
      val x3 = x1 + 1; val x5 = println(x3)
    () } else { () } ()
  }
}
```

State?

```
class RepFoldOp[A, B](f: (Rep[A], Rep[B]) => Rep[B], z: Rep[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 }
}
```

```
fold[Int]({(x, y) => x + y}, 1)
```

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

```
(1, 2, 3, 4) => (2, 3, 4, 5) instead of (2, 4, 7, 11)
```

Stateful RepStreamOps

```
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 = ...
}
```

The correct fold

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

⇩

```
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); ()
  }
}
```

RepStreamOp → StreamOp

```
trait RepStreamCompile extends RepStreamOpsExp with ScalaCompile { self =>
  val codegen = new ScalaGenRepStreamOps {
    val IR: self.type = self

    def emitSourceStream[T: Manifest](s: RepStreamOp[T], className: String,
        out: PrintWriter): Unit = {
      emitSource(s.onData _, className + "$onData", out)
      emitSource({x: Rep[Unit] => s.flush}, className + "$flush", out)
    }
  }

  def compileStream[T: Manifest](s: RepStreamOp[T]): StreamOp[T] = {
    new StreamOp[T] {
      val onDataFun: (T => Unit) = compile(s.onData _)
      val flushFun: (Unit => Unit) = compile({x: Rep[Unit] => s.flush})

      def onData(data: T) = onDataFun(data)
      def flush = flushFun()
    }
  }
}
```

GroupBy

- push the full HashMap
- push lists (groups)
- push elements (could then aggregate into Lists again)

- store state in groupBy
- lifted RepStreamOps?
- compile?

WindowJoin

J. Teubner, R. Mueller: How Soccer Players Would do Stream Joins (SIGMOD 2011)

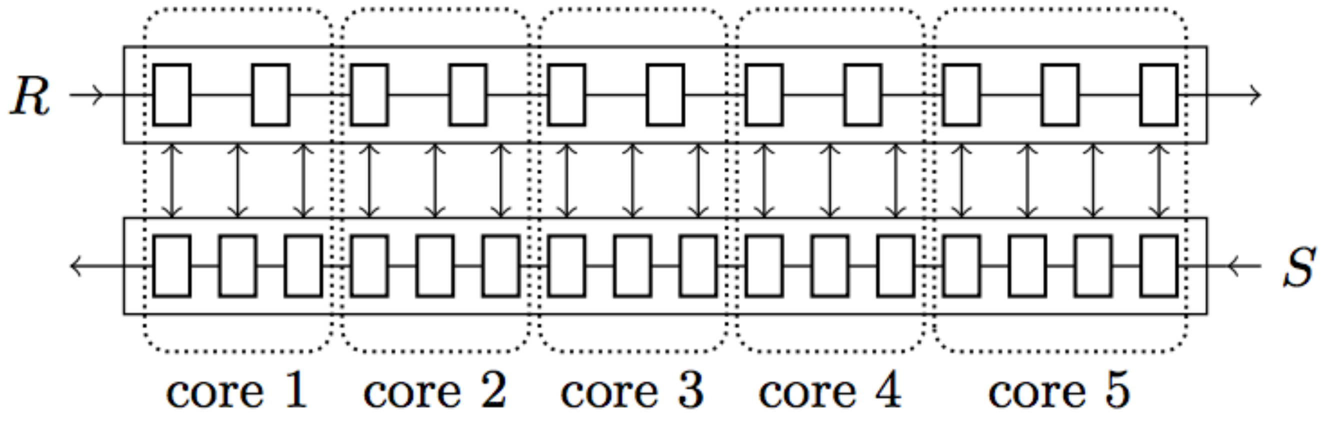

Figure 5: Parallelized handshake join evaluation. Each compute core processes one segment of both windows and performs all comparisons locally.

Future work

Now:

- publish a paper
- think more about groupBy
- properly encapsulate Cell[T]
- Benchmarks

Further ideas:

- check DBToaster as use case
- better WindowJoin

Conclusion

Thank you for your attention!

Contact: vera.salvisberg@epfl.ch
Code: https://github.com/vsalvis/DslStreams

Questions?