Fold-Based Fusion as a Library

A Generative Programming Pearl

Manohar Jonnalagedda, Sandro Stucki, EPFL

Scala ‘15, Portland, June 13 2015
An Example

```scala
val people2Movies: List[(String, List[String])] = List(
  ("Sébastien", List("Hunger Games", "Silver Linings Playbook")),
  ("Eugene", List("Gattaca", "Inside Man")),
  ("Hubert", List("Silver Linings Playbook", "Lost in Translation")),
  ("Sandro", List("Lost in Translation", "The Matrix", "Pulp Fiction")),
  ("Heather", List("Django Unchained", "Tropic Thunder", "Pulp Fiction")),
  ...
)

Question: How many people like each movie?
The Scala Way

def movieCount(people2Movies: List[(String, List[String])]): Map[String, Int] = {
  val flattened = for {
    (person, movies) <- people2Movies
    movie <- movies
  } yield (person, movie)
  val grouped = flattened groupBy (_._2)
  grouped map { case (movie, ps) => (movie, ps.size) }
}
def movieCount2(people2Movies: List[(String, List[String])]): Map[String, Int] = {
  var tmpList = people2Movies;
  val tmpRes: Map[String, Int] = Map.empty
  while (!tmpList.isEmpty) {
    val hd = tmpList.head; var movies = hd._2
    while (!movies.isEmpty) {
      val movie = movies.head
      if (tmpRes.contains(movie)) {
        tmpRes(movie) += 1
      } else tmpRes.update(movie, 1)
      movies = movies.tail
    }
    tmpList = tmpList.tail
  }
  tmpRes
}
Fusion (Deforestation)

- movieCount (more readable) → movieCount2 (no intermediate structures)
- Desirable properties of a fusion algorithm
  - should deforest as many operations as possible.
  - should be simple, elegant even.
Fusion in the Large

- Haskell
  - use built-in fusion algorithms.
  - use rewrite rule system.

- Scala
  - Scala Blitz
  - The Dotty Linker
In this Presentation

- Fusion as a Library (aka let’s build it ourselves)
  - Fold-Based Fusion, as powerful as foldr/build Fusion.
  - Applies to producers.
  - Also works for partitioning and grouping functions.
The Gist

● Convert Data Structures to their CPS-encoded versions
  ○ composition over structures -> composition over functions

  FoldLeft

● Partially evaluate function composition -> deforestation

  Lightweight Modular Staging
FoldLeft

def foldLeft[A, S](ls: List[A])(z: S, comb: (S, A) => S): S = ls match {
  case Nil => z
  case x :: xs => foldLeft(xs)(comb(z, x), comb)
}

  Nil,
  (acc, elem) => acc ++ f(elem)
)
FoldLeft

```scala
def filter(ls: List[A], p: A => Boolean) = foldLeft[A, List[A]](ls)(
    Nil,
    (acc, elem) => if (p(elem)) acc :+ elem else acc
)

def flatMap[A, B](ls: List[A], f: A => List[B]) = foldLeft[A, List[B]](ls)(
    Nil,
    (acc, elem) => foldLeft[B, List[B]](f(elem))(acc,
        (acc2, elem) => acc2 :+ elem
    )
)
```
The Essence of FoldLeft

\[
\text{foldLeft}[A, S]: \text{List}[A] \Rightarrow ((S, (S, A) \Rightarrow S) \Rightarrow S)
\]
Abstracting over CPS-Encoded Lists

//foldLeft[A, S]: List[A] => ((S, (S, A) => S) => S)

type Comb[A, S] = (S, A) => S

abstract class CPSList[A] { self =>
    def apply[S](z: S, comb: Comb[A, S]): S
    ...
}

The API of CPSList

//foldLeft[A, S]: List[A] => ((S, (S, A) => S) => S)
abstract class CPSList[A] { self =>

  ...
  def map[B](f: A => B): CPSList[B] = ...
  def filter(p: A => Boolean): CPSList[A] = ...
  def flatMap(f: A => CPSList[B]): CPSList[B] = ...
}

object CPSList {
  def fromList[A](ls: List[A]): CPSList[A] = ...
  def fromRange(a: Int, b: Int): CPSList[Int] = ...
}
The API of CPSList

  Nil,
  (acc, elem) => acc ::: f(elem)
)

abstract class CPSList[A] { self =>
  ...
  def map[B](f: A => B) = new CPSList[B] {
    def apply[S](z: S, comb: Comb[B, S]) = self.apply(
      z,
      (acc: S, elem: A) => comb(acc, f(elem))
    )
    }
  ...
}
Using CPSList

```scala
def listExample(a: Int, b: Int) = (a to b).toList
  .flatMap(i => (1 to i).toList)
  .filter(_ % 2 == 1)
  .map(_ * 3).sum

def cpsListExample(a: Int, b: Int) = {
  val pipeline =
    CPSList.fromRange(a, b).flatMap(i => CPSList.fromRange(1 to i))
    .filter(_ % 2 == 1)
    .map(_ * 3)
  pipeline.apply[Int](0, (acc, x) => acc + x)
}
```

fold only applied at the very end
Welcome to Part II

- Convert Data Structures to their CPS-encoded versions
  - composition over structures $\rightarrow$ composition over functions

  FoldLeft

- Partially evaluate function composition $\rightarrow$ deforestation

  Lightweight Modular Staging
Partial Evaluation and Staging

- Partial evaluation
  - pre-evaluate parts of a program
  - residual program is specialized → better performance

- Staging, aka. Multi-Stage Programming
  - Separate parts of the program in terms of evaluation
  - some parts are executed “now”
  - other parts are delayed to the next stage
  - => use staging for controlled partial evaluation
def cpsListExample(a: Int, b: Int) = {
  val pipeline =
      CPSList.fromRange(a, b).flatMap(i => CPSList.fromRange(1 to i))
      .filter(_ % 2 == 1)
      .map(_ * 3)
  pipeline.apply[Int](0, (acc, x) => acc + x)
}
def cpsListExample(a: Int, b: Int) = {
  val pipeline$1 =
      CPSList.fromRange(a, b).flatMap(i => CPSList.fromRange(1 to i))
        .filter(_ % 2 == 1)
  pipeline$1.apply[Int](0, (acc, x) => acc + x * 3)
}
def cpsListExample(a: Int, b: Int) = {
    val pipeline$1$2 =
        CPSList.fromRange(a, b).flatMap(i => CPSList.fromRange(1 to i))
    pipeline$1$2.apply[Int](
        0,
        (acc, x) => if (x % 2 == 1) acc + x * 3 else acc
    )
}
def cpsListExample(a: Int, b: Int) = {
  val pipeline$1$2$3 = CPSList.fromRange(a, b)
  pipeline$1$2$3.apply[Int](
    0,
    (acc, x) => CPSList.fromRange(1 to x).apply[Int](
      acc,
      (innerAcc, y) => if (y % 2 == 1) innerAcc + y * 3
        else innerAcc
    )
  )
}
def cpsListExample(a: Int, b: Int) = {
  @tailrec
def loop(a1: Int, b1: Int, tmpRes: Int) =
    if (a1 > b1) tmpRes
    else loop(a1 + 1, b1, innerLoop(1, a, tmpRes))

  @tailrec
def innerLoop(i1: Int, i2: Int, tmpRes: Int) =
    if (i1 > i2) tmpRes
    else innerLoop(i1 + 1, i2,
                  if (i1 % 2 == 1) tmpRes + i1 * 3 else tmpRes
               )
  loop(a, b, 0)
}

Partially Evaluating CPSList
def cpsListExample(a: Int, b: Int) = {
  var tmpRes: Int = 0; var i = a
  while (i <= b) {
    var j = 1
    while (j <= i) {
      if ((j % 2) == 1) { tmpRes += j * 3 }
      j += 1
    }
    i += 1
  }
  tmpRes
}
The Punchline

- Convert Data Structures to their CPS-encoded versions
  - composition over structures $\rightarrow$ composition over functions

- Partially evaluate function composition $\rightarrow$ deforestation
Partial Evaluation with LMS

```
def add3(a: Int, b: Int, c: Rep[Int]) =
a + b + c
```

```
def add$1$2$c(c: Int) =
3 + c
```

```
def add3(a: Int, b: Int, c: Int) =
a + b + c
```

```
add3(1, 2, x)
```

Expression in the next stage

Adding Rep types

Executed at staging time
Constant in the next stage

```
add3(1, 2, 3)
```

Partial evaluation/Code generation

Evaluation of generated code

```
add$1$2$c(3)
```

Direct evaluation

```
6
```
Partial Evaluation with LMS: Functions

```scala
def apply(a: Int, f: Int => Int) = f(a)
apply(2, _ * 3) // 6

def apply(a: Rep[Int], f: Rep[Int] => Rep[Int]) = f(a)
apply(x, _ * 3)
```

Adding Rep types

```scala
def apply(a: Int, f: Int => Int) = f(a)
apply(a$3(2))
```

Executed at staging time
Constant in the next stage

Expression in the next stage

Partial evaluation/Code generation

Evaluation of generated code
LMS

User-written code, may contain Rep types

LMS runtime code generation

Generated/optimized code.
Staging CPSList

\[\text{foldLeft}[A, S]: \text{List}[A] \Rightarrow ((S, (S, A) \Rightarrow S) \Rightarrow S)\]
Staging CPSList

\[
\text{stagedFoldLeft}[A, S] : \text{Rep[List}[A]] \Rightarrow \\
((\text{Rep}[S], (\text{Rep}[S], \text{Rep}[A]) \Rightarrow \text{Rep}[S]) \Rightarrow \text{Rep}[S])
\]
Staging CPSList

type Comb[A, S] = (Rep[S], Rep[A]) => Rep[S]

abstract class CPSList[A] { self =>
    def apply[S](z: Rep[S], comb: Comb[A, S]): Rep[S]
        ...
}

The API of Staged CPSList

abstract class CPSList[A] { self =>
  ...
  def map[B](f: Rep[A] => Rep[B]): CPSList[B] = ...
  def filter(p: Rep[A] => Rep[Boolean]): CPSList[A] = ...
  def flatMap(f: Rep[A] => CPSList[B]): CPSList[B] = ...
}

object CPSList {
  def fromList[A](ls: Rep[List[A]]): CPSList[A] = ...
  def fromRange(a: Rep[Int], b: Rep[Int]): CPSList[Int] = ...
}
The Rabbit out of the Hat

- Convert Data Structures to their CPS-encoded versions
  - composition over structures $\rightarrow$ composition over functions

- Partially evaluate function composition $\rightarrow$ deforestation

- Multiple Producers
Multiple Element Producers

● So far: API contains only single element producers
● Next, partitioning and grouping:
  ○ Produce multiple elements.
  ○ We look at partitioning here.
  ○ Grouping: in the paper/talk to me later.
The Partition Function

```scala
def partition[A](ls: List[A], p: A => Boolean): (List[A], List[A]) = 
  foldLeft[A, (List[A], List[A])](ls)(
    (Nil, Nil), {
      case ((trues, falses), elem) =>
        if (p(elem)) (trues ++ List(elem), falses)
        else (trues, falses ++ List(elem))
    }
  )

val myList: List[Int] = ...
val (evens, odds) = partition(myList, (x: Int) => x % 2 == 0)
(evens map (_ * 2), odds map (_ * 3))
```
Staged Partition, a Naive Attempt

//as a method on CPSList

def partition(p: Rep[A] => Rep[Boolean]): (CPSList[A], CPSList[A]) = {
  val trues  = this filter p
  val falses = this filter (a => !p(a))
  (trues, falses)
}
def partitionE[A](ls: List[A], p: A => Boolean): List[Either[A, A]] =
  ls map { elem => if (p(elem)) Left(elem) else Right(elem) }

val myList: List[Int] = ...
val partitioned = partitionE(myList, (x: Int) => x % 2 == 0)
val mapped = partitioned map {
  case Left(x) => Left(x * 2)
  case Right(x) => Right(x * 3)
}

foldLeft[Either[Int, Int], (List[Int], List[Int])](mapped)(
  (Nil, Nil), {
    case ((trues, falses), elem) =>
      elem.fold(x => (trues ++ List(x), falses), x => (trues, falses ++ List(x)))
  })
Staged Partition, Bis

//as methods on CPSList

def partitionBis(p: Rep[A] => Rep[Boolean]): CPSList[Either[A, A]] =
    this map { elem =>
        if (p(elem)) left[A, A](elem)
        else right[A, A](elem)
    }

Rep[Either] means boxes in generated code!
Remembering the Gist: CPSEither

abstract class CPSEither[A, B] {
    def apply[X](lf: A => X, rf: B => X): X
}

----> LMS ---->
abstract class CPSEither[A, B] {
    def apply[X](
        lf: Rep[A] => Rep[X],
        rf: Rep[B] => Rep[X]
    ): Rep[X]
...
}
More in the Paper on

- Grouping functions
  - CPS encoding of tuples
- LMS-specific implementation details
  - Code generation for conditional expressions
## The Ecosystem of Fusion Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Foldr/build, Staged CPSList</th>
<th>Unfoldr/destroy</th>
<th>Stream fusion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Producers</strong></td>
<td>✓</td>
<td>✘</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Consumers</strong></td>
<td>✘</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Details</strong></td>
<td>cannot handle zip-like functions</td>
<td>issues with filter, flatMap</td>
<td>fusion algorithm more involved, esp. for flatMap</td>
</tr>
</tbody>
</table>
Thank you!

https://github.com/manojo/staged-fold-fusion/
Staged CPSList

foldLeft[A, S]: List[A] => (S, (S, A) => S) => S

def fromList[A](ls: Rep[List[A]]) = new CPSList[A] {
  def apply[S](z: Rep[S], comb: Comb[A, S]): Rep[S] = {
    var tmpList = ls
    var tmp = z
    while (!tmpList.isEmpty) {
      tmp = comb(tmp, tmpList.head)
      tmpList = tmpList.tail
    }
    tmp
  }
}
The Staged CPSList API

//as methods of CPSList

  def apply[S](z: Rep[S], comb: Comb[B, S]) = self.apply(
    z,
    (acc: Rep[S], elem: Rep[A]) => comb(acc, f(elem))
  )
}
The Staged CPSList API

//as methods of CPSList
  def apply[S](z: Rep[S], comb: Comb[A, S]) = self.apply(
    z,
    (acc: Rep[S], elem: Rep[A]) =>
      if (p(elem)) comb(acc, elem) else acc
  )
}
The Staged CPSList API

//as methods of CPSList

  def apply[S](z: Rep[S], comb: Comb[B, S]) = self.apply(
    z,
    (acc: Rep[S], elem: Rep[A]) => f(elem)(acc, comb)
  )
}

An Example

def foldLeftExample(a: Rep[Int], b: Rep[Int]): Rep[Int] = {
    val fld = CPSList.fromRange(a, b)
    val flatMapped = fld flatMap {
        i => CPSList.fromRange(1, i)
    }
    val filtered = flatMapped filter (_ % 2 == 1)
    filtered.map(_ * 3).apply[Int](
        0, (acc, x) => acc + x
    )
}
An Example

def generatedFunction(x0:Int, x1:Int): Int = {
    var x2: Int = x0; var x3: Int = 0
    while (x2 <= x1) {
        val x7 = x3
        val x8 = x2
        var x9: Int = 1
        var x10: Int = x7
        ...
        val x26 = x10; x3 = x26
        val x28 = x8 + 1; x2 = x28
    }
    val x32 = x3
    x32
}

while (x9 <= x8) {
    val x14 = x10; val x15 = x9
    val x16 = x15 % 2; val x17 = x16 == 1
    val x20 = if (x17) {
        val x18 = x15 * 3
        val x19 = x14 + x18
        x19
    } else {
        x14
    }
    x10 = x20
    val x22 = x15 + 1
    x9 = x22
}
Staging = Multi-Stage Programming

- Separate parts of the program in terms of evaluation
  - some parts are executed “now”
  - other parts are delayed to the next stage
- Related concept: partial evaluation
  - pre-evaluate parts of a program
  - residual program is specialized -> better performance
Staging (LMS)

Rep[T => U]


Rep[T] => Rep[U]

Unstaged function on staged types. Application inlines body of function. Generated code contains no function call.