Fold-Based Fusion as a Library

A Generative Programming Pearl

Manohar Jonnalagedda Sandro Stucki

LAMP, EPFL {first.last}@epfl.ch

Abstract

Fusion is a program optimisation technique commonly implemented using special-purpose compiler support. In this paper we present an alternative approach, implementing fold-based fusion as a standalone library. We use staging to compose operations on folds; the operations are partially evaluated away, yielding code that does not construct unnecessary intermediate data structures. The technique extends to partitioning and grouping of collections.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Code generation, Optimisation, Parsing

Keywords fusion; fold; multi-stage programming

1. Introduction

Suppose you are given a list of people, along with a list of movies each of these people like. If you want to find out how many people like each movie, here is a Scala snippet to do the job:

```
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) }
  }
```

The function uses two intermediate data structures, `flattened` and `grouped`. These data structures are helpful in organising the program and making it more readable. On the other hand, their allocation and construction incurs a significant memory and processing overhead. Yet it is possible to implement the `movieCount` function without creating any intermediate structures. The following implementation is arguably harder to read, but more efficient:

```
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 is a program transformation that converts functions written in a `movieCount` style to efficient equivalents in the `movieCount2` style. Its goal is to avoid the creation of costly intermediate data structures. Fusion has been extensively studied, both theoretically [6] and in practice [2, 5, 13].

Practical implementations of this technique tend to rely on an optimising compiler for a pure, functional language. In non-pure languages, it is more difficult to implement fusion as part of the compiler, due to the possible presence of side-effects, open recursion in datatypes, virtual method dispatch, etc. There are however many pure, functional subdomains in such languages that could greatly benefit from fusion. Examples for such subdomains include collection libraries and query-like languages. Essentially, programs that process data through "pipelines" of operations are amenable to fusion.

In this paper, we present fold-based fusion as a library. This decouples the optimisation from an underlying compiler, making it portable, and readily applicable to different contexts. Our implementation hinges on combining two insights:

1. By CPS-encoding data, we can reduce fusion of arbitrary data types to fusion of functions.
2. Multi-staged programming [15] allows us to partially evaluate function composition, thus effectively achieving fusion of functions through partial evaluation.

Contributions. This suggests a generative programming [9] approach to fusion:

- We present an API for staged, CPS-encoded lists (Section 3). Staging (Section 2) is used as a means to systematically separate function composition from data processing . A programmer using our library has the impression that he is composing operations over folds. In fact, he/she composes operations over *code generators of folds*. This composition is partially evaluated away at staging time, yielding code that contains no intermediate data structures. Our fusion technique remains as powerful as `foldr/build` fusion [5]: it does well on functions that are "good producers".
- Some producer functions are less good than others: they produce multiple outputs. Partitioning and grouping fall under this category. We discuss variants of these functions that are easier to fuse (Section 4).
- These variants introduce extra boxes around data in order to preserve operating under a single pipeline. We present a technique to systematically eliminate them. Once again, the key is to CPS-encode the data representations of the boxes, and stage these representations. We explain the technique for the `Either` type in particular (Section 5).

By embracing generative programming as a paradigm and combining it with functional APIs, we get an implementation that has a library look-and-feel.

2. Staging

We implement our fusion library using the Lightweight Modular Staging (LMS) framework [10]. This section provides a short overview of the framework and necessary background on the partial evaluation techniques used in Sections 3–5.

2.1 Partial evaluation and multi-stage programming

Partial evaluation [3] is a technique used primarily to perform program optimisation. In a program receiving static and dynamic inputs, computations over statically known values are evaluated away, thereby *specialising* the program for that particular static input.

A closely related concept is multi-stage programming or staging [15], a form of generative programming. In a multi-staged program, one explicitly specifies which parts of the program are to be evaluated at the current stage, and which parts should be evaluated at a later stage. Running a staged program generates a new program, where current stage computations have been evaluated away. MSP can therefore be used to achieve controlled partial evaluation.

2.2 LMS

LMS is a staging/runtime code generation framework written in Scala. The evaluation of expressions is controlled through the use of a special abstract type `Rep[T]`:

- an expression of type `T` evaluates to a constant of type `T` in the generated code,
- an expression of type `Rep[T]` generates code for an expression of type `T`.

Figure 1 illustrates this principle. Starting from a program as in the bottom-left corner, a programmer adds `Rep` types, as in the top-left corner. The LMS framework will run this program, yielding later-stage code (top-right corner). Only when this code is executed do we get the final result of the program. Note that when we compose expressions of type `Rep[T]`, we *compose code generators*. Scala's type system and implicits allow LMS programs to look essentially like their unstaged counterparts.

Staged Functions. A key concept in LMS is the distinction between the function types `Rep[T => U]` and `Rep[T] => Rep[U]`. The former type is that of a staged function, i.e. it will generate a function in later-stage code. The latter type is that of an unstaged function on staged types. Applying it to an input of type `Rep[T]` expands the function definition at the call site, effectively inlining it.

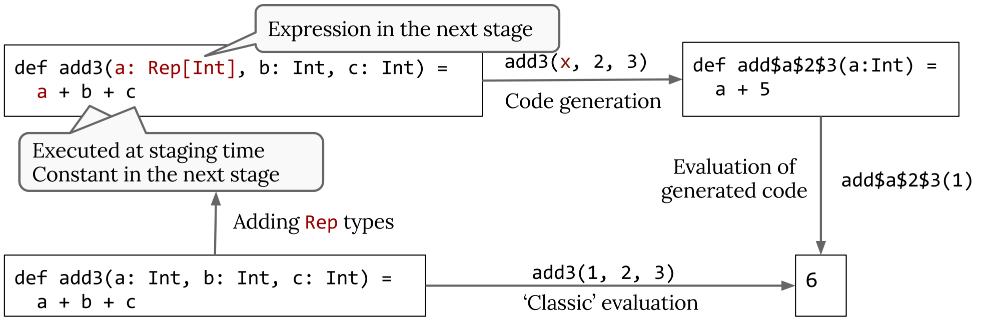
Unstaged functions play a key role in the design of staged libraries. Using them, we get inlining for free, and avoid allocating unnecessary closures. This idea extends to higher-order functions, which may take unstaged functions as parameters.

The LMS intermediate representation. Every instance of the abstract type `Rep[T]` corresponds to a concrete datatype, which can be pattern matched against and rewritten. The collection of such datatypes forms the *LMS intermediate representation* (IR). A common use case for rewrites is optimisations. For example, a conditional expression where the condition is constant may be replaced by one of its branches.

The core LMS library defines intermediate nodes and rewrite rules for many common programming constructs, such as conditionals, Boolean expressions, arithmetic expressions and list operations. These building blocks can be used out of the box in order to build more complex code generators [12].

3. Staging FoldLeft

Having introduced Lightweight Modular Staging, we now move on to the main topic of this paper, which is to achieve fusion on operations over collections. For simplicity, we restrict ourselves to lists. We consider a type of fusion which is applicable to operations over lists that are expressible as folds, i.e. fold-based fusion.



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

```
def add$a$2$3(a:Int) =
  a + 5
```

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

Figure 1. Staging in LMS

3.1 FoldLeft

Many operations on lists can be implemented in terms of the generic fold function [5]. For lists, there are two variants of the fold operation, `foldLeft` and `foldRight`. The two operations are equivalent in that one can be implemented using the other. We choose `foldLeft`: we will see later in this section why this representation benefits us more.

The `foldLeft` function on lists can be implemented as follows:

```
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)
  }
```

It takes a zero (or initial) element of type `S`, and returns this element if the input list is empty. If the list contains some elements, they are recursively combined with the element `z` using the binary operator `comb`. The elements are combined to the left, hence the name of the function.

As mentioned above, various operations on lists can be implemented using `foldLeft`. We defer the presentation of the full API to Section 3.3, and give an example implementation of the `map` function instead:

```
def map[A, B](ls: List[A], f: A => B): List[B] =
  foldLeft[A, List[B]](ls)(
    Nil,
    (acc, elem) => acc :+ f(elem)
  )
```

Starting with an empty list, the combination function simply appends to the accumulator the results of applying `f` to the elements of the input list.

CPS-encoded lists. Consider the type signature of the `foldLeft` function:

```
List[A] => (S, (S, A) => S) => S
```

The signature tells us that, given a list over any type `A`, `foldLeft` returns a function that will fold the elements of that list into a structure of some type `S`. The type of this function turns out to be the CPS encoding (also known as the Church encoding) of lists, or equivalently the list functor [8]:

```
type FoldLeft[A, S] = (S, (S, A) => S) => S
```

Here, `S` denotes the eventual result type of operations over the list. For instance in the above `map` example, `S = List[A]`. In essence, `foldLeft` maps plain lists to CPS-encoded lists.

3.2 Foldleft, staged

Having captured the essence of fold with the type alias `FoldLeft`, we can now stage this representation. Following the ideas outlined in Section 2, we come up with the following type alias:

```
type FoldLeft[A, S] =
  (Rep[S], (Rep[S], Rep[A]) => Rep[S]) => Rep[S]
```

Note that the name is deliberately overloaded. For the rest of the paper, unless explicitly mentioned, `FoldLeft` refers to the staged version. As promised, we use unstaged functions.

Figure 2 shows an implementation of staged `FoldLeft` in LMS. The enclosing trait `FoldLefts` mixes in some of LMS' building blocks which help in composing code generators [12]. These are the only blocks required for `FoldLeft`. In particular, we want to be able to write a bit of mutable code (`LiftVariables`) and while loops (`While`). The `Manifest` annotation on polymorphic types is specific to code generation.

`FoldLeft` is not a type alias, but an abstract class now. This way we can add methods to its API. The type parameter `A` represents the type of elements that pass through it. Every instance of `FoldLeft` must implement an `apply` method, corresponding to the application of fold. As explained above, the type parameter `S` for this method corresponds to the eventual structure resulting from the fold.

We create a `FoldLeft` over a list with the `fromList` function. Since `FoldLeft` corresponds to the return type of the `foldLeft` function on lists, `fromList` is a staged code generator for `foldLeft`. Here we choose an implementation using loops instead of recursion. This is because the target languages for our code generation (Scala, Java or C) are better at executing while loops than recursive functions. This also explains our choice of `foldLeft`: contrary to `foldRight`, it can be imple-

```
trait FoldLefts
    extends ListOps
    with IfThenElse
    with BooleanOps
    with Variables
    with OrderingOps
    with NumericOps
    with PrimitiveOps
    with LiftVariables
    with While {

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

  abstract class FoldLeft[A: Manifest] { self =>
    def apply[S: Manifest](z: Rep[S], comb: Comb[A, S]): Rep[S]
    //operations on foldleft go here
  }
}

//companion object
object FoldLeft {
  //create a fold from a list
  def fromList[A: Manifest](ls: Rep[List[A]]) =
    new FoldLeft[A] {
      def apply[S: Manifest](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
        }
    }
  ...
}
```

Figure 2. FoldLeft as a staged abstraction

mented in a tail-recursive manner, hence easily written as a low-level loop.

Note that `fromList` takes as parameter a `Rep[List[A]]`, and not a `List[Rep[A]]`. Indeed, the input list to a pipeline of folds is not usually known statically.

3.3 The API of staged FoldLeft

We now extend our staged `FoldLeft` implementation by adding a list-like API. Note that these methods can be added to an unstaged `FoldLeft` as well. The only difference is the use of staged types and unstaged functions. Figure 3 shows the API. We add `fromRange` to the companion object, which creates a `FoldLeft` from an integer interval. The rest of the API consists of the usual suspects, `map`, `filter`, `flatMap` and `concat`. We remark that:

```
//as methods of FoldLeft
def map[B: Manifest](f: Rep[A] => Rep[B]) =
  new FoldLeft[B] {
    def apply[S: Manifest](z: Rep[S], comb: Comb[B, S]) =
      self.apply(
        z,
        (acc: Rep[S], elem: Rep[A]) => comb(acc, f(elem))
      )
  }

def filter(p: Rep[A] => Rep[Boolean]) =
  new FoldLeft[A] {
    def apply[S: Manifest](z: Rep[S], comb: Comb[A, S]) =
      self.apply(
        z,
        (acc: Rep[S], elem: Rep[A]) =>
          if (p(elem)) comb(acc, elem) else acc
      )
  }

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

def concat(that: FoldLeft[A]) = new FoldLeft[A] {
  def apply[S: Manifest](z: Rep[S], comb: Comb[A, S]) = {
    val folded: Rep[S] = self.apply(z, comb)
    that.apply(folded, comb)
  }
}

//in the companion object
def fromRange(a: Rep[Int], b: Rep[Int]) =
  new FoldLeft[Int] {
    def apply[S: Manifest](z: Rep[S], comb: Comb[Int, S]) = {
      var tmpInt = a
      var tmp = z
      while (tmpInt <= b) {
        tmp = comb(tmp, tmpInt)
        tmpInt = tmpInt + 1
      }
      tmp
    }
  }
```

Figure 3. The API of staged FoldLeft

```
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
    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
    }
    val x26 = x10
    x3 = x26
    val x28 = x8 + 1
    x2 = x28
  }
  val x32 = x3
  x32
}
```

Figure 4. Example code generated by LMS.

- Most of the operations take unstaged functions over staged types as arguments. The body of these functions is inlined at application site as a result.
- The type of the function argument `f` of `flatMap` deserves some elaboration. Expanding the type of `FoldLeft`, we get the following type for `f`:

  ```
  f: Rep[A] => (Rep[S], Comb[B, S]) => Rep[S]
  ```

  which is a curried, unstaged function. By fully applying this function, we inline not only the body of `f`, but also the body of the resulting `FoldLeft`. This way, we avoid generating code for an intermediate collection. The same holds for `concat`.
- The function passed to `flatMap` must return `FoldLeft`. If this `FoldLeft` is created from a call to `fromList`, an intermediate list will be generated as well. A programmer must therefore be careful how to create this `FoldLeft`.

A code generation example As mentioned in Section 2, LMS takes as input a staged program, and generates a later-stage program. Consider the following example that uses `FoldLeft`:

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

Given an integer interval, it creates nested intervals. It then sums all odd elements of the nested intervals, after having multiplied them by 3. Note that in the `flatMap` call, we pass a function that creates a fold from an interval, rather than from a list. Running LMS will partially evaluate the staged `FoldLeft` away, yielding code as in Figure 4. As we can see, we are left with two nested while loops, exactly what we wished for.

The power of staged `FoldLeft`. We now have a library over a staged fold abstraction, which enables us to write pipelines of operations over lists. Through partial evaluation, we generate code that is devoid of intermediate data structures. The main difficulty consisted in identifying the correct types for unstaged function arguments.

It is natural to wonder how many of the common operations over lists are fusible by this technique in practice. Our staged `FoldLeft`, being purely fold based, is as powerful as *foldr/build* fusion [5]. Indeed, we face the same problem with zips and other functions that consume multiple inputs. Fold-based fusion works well for operations that act as "good producers".

4. Partitioning and Grouping

In the previous section, we only considered list operations that produce exactly one list as their result. It is not much of a surprise that such functions should be amenable to fusion since their composition will always result in "straight pipelines", i.e. functions which again take lists to lists. This is sometimes referred to as vertical fusion.

In this section we turn to operations that produce multiple outputs, and hence allow us to build forked pipelines. The main challenge consists in keeping all operations in the same pipeline, while avoiding the introduction of intermediate data structures to do so. This is also known as horizontal fusion. We start with the `partition` function.

4.1 Partition

The `partition` function on lists takes a list and a predicate, and returns two lists, one containing the elements satisfying the predicate, and the other containing those that do not. We can implement this function using `foldLeft` as defined in Section 3.1:

```
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))
    }
  )
```

The initial element is a pair of empty lists. Based on the predicate, we add each element of the input list to either the first of the second accumulating list.

Here is an example usage of `partition`:

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

In the context of fusion, we naturally want to avoid creating the `evens` and `odds` lists.

A naive attempt. One possibility for implementing `partition` on `FoldLeft` is to have it return two separate `FoldLefts`:

```
//as a method on FoldLeft
def partition(p: Rep[A] => Rep[Boolean]):
    (FoldLeft[A], FoldLeft[A]) = {
  val trues = this filter p
  val falses = this filter (a => !p(a))
  (trues, falses)
}
```

This looks great, because though we create a pair, it is unstaged and so is partially evaluated away. Moreover, we can access both `FoldLefts` separately and further construct their pipelines separately.

Unfortunately, if both `trues` and `falses` are used later on, code for two separate traversals over the entire pipeline will be generated, which defeats the point of fusion. It is preferable to have a single traversal.

Partition with Either. If our objective is to generate a single traversal, we must fix the return type for `partition` to be `FoldLeft`, our current abstraction for loops. This particular `FoldLeft` does not see elements of type `A` anymore, but elements that have either passed a predicate, or not. The `Either` type captures this notion very well: instances of `Left` represent elements satisfying the predicate, instances of `Right` represent elements that do not. We can rewrite the example above as follows:

```
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))
    )
  }
)
```

The `partitionE` function is simply an application of the `map` function, turning an element of type `A` into an element of type `Either[A, A]`. It has the effect of *delaying* the creation of two separate lists to a later application of `foldLeft`. Between the final application and the partition point, we use the `map` function on `Either` to thread computations through to the actual values.

Note that `eventually`, we are left with no option but to fork the pipeline into two lists, through a final call to `foldLeft`. Here, the combination operation concatenates elements to the resulting lists through the use of the `fold` function on `Either`.

The implementation of `partitionE` above gives us a way to implement the same for staged `FoldLeft`. Figure 5 shows the code. The functions `left` and `right` create instances of `Rep[Either]`.

The reader will surely object to this implementation. We have not really eliminated intermediate data structures. Rather, we have created new ones, in the form of instances of `Rep[Either]`. The insight is that we *know* exactly what type of boxes we create. We discuss shortly how to eliminate them (Section 5). Before that, we discuss another multiple output producer function, `groupBy`.

```
//as methods of the FoldLeft class
def partitionBis(p: Rep[A] => Rep[Boolean]):
    FoldLeft[Either[A, A]] = this map { elem =>
  if (p(elem)) left[A, A](elem) else right[A, A](elem)
}

def groupWith[K: Manifest](f: Rep[A] => Rep[K]):
  FoldLeft[(K, A)] = this map { elem => (f(elem), elem) }
```

Figure 5. The partition and groupWith methods on FoldLeft

4.2 GroupBy

The partition function on `FoldLeft` enables us to write pipelines so that no intermediate lists are created, and the single traversal requirement is met. We now focus our attention on a cousin of `partition`'s, `groupBy`.

While partitioning splits a list into two groups, `groupBy` partitions a list into possibly many groups. This operation is also particularly interesting because it is a common query operation. It is of course used in query languages, but it is also not uncommon in spreadsheet-like languages to visualise results better. Recall the example in Section 1, where we group movies by people who like them, and then count the number of people per group.

For lists, the `groupBy` function can be implemented as follows, once again using `foldLeft`:

```
def groupBy[A, K](ls: List[A], f: A => K): Map[K, List[A]] =
  foldLeft[A, Map[K, List[A]]](ls)(
    Map.empty[K, List[A]],
    {case (dict, elem) =>
      val k = f(elem)
      if (dict.contains(k))
        dict + ((k, dict(k) ++ List(elem)))
      else
        dict + ((k, List(elem)))
    }
  )
```

It takes an input list, and a function `f` that attributes a key to a value. It returns a collection of key-value pairs, where the value is itself a collection of values from the input list `ls`. The initial element passed to the fold is an empty map. The combination operator adds a new key-value pair to the map if the key has not been created yet. Otherwise, it appends the element to the pre-existing list.

We can reimplement the example from the introduction using the above implementation of `groupBy`:

```
def movieCount(people2Movies: List[(String, List[String])]):
  Map[String, Int] = {
    val flattened = for {
      (person, movies) <- people2Movies
      movie            <- movies
    } yield (person, movie)
    val grouped = groupBy[(String, String), String](
      flattened, _._2
    )
    grouped map { case (movie, ls) => (movie, ls.size) }
  }
```

Note that we use a map function on `HashMap` after the call to `groupBy`. Once again, in terms of fusion, we would like to avoid creating the intermediate `HashMap[Int, List[Int]]`. One possibility for the above example is to implement a specific `reduceBy` function that takes an extra reduction function and applies it. Many collection libraries do indeed contain this alternative. We may however want to first group elements, perform group-specific operations on the values, and then reduce them. In which case a `reduceBy` will not suffice.

Delaying the application of `FoldLeft`. As in the case for partition, the key idea is to keep everything on a single fold pipeline for as long as possible. To achieve this, we once again resort to introducing an extra box type, through the use of a function named `groupWith`. This function is shown in Figure 5. The result of applying a `groupWith` is a `FoldLeft` over key-value pairs. Values from the input fold are simply tagged with their group, and sent further down the pipeline. The above grouping example can be written for staged `FoldLeft` as below:

```
def repMovieCount(
  people2Movies: Rep[List[(String, List[String])]])
  : Rep[HashMap[String, Int]] = {
    val fld = FoldLeft.fromList[(String, List[String])](
      people2Movies)

    val flattened: FoldLeft[(String, String)] = for {
      elem  <- fld
      movie <- FoldLeft.fromList[String](elem._2)
    } yield (elem._1, movie)

    val grouped = flattened groupWith { elem => elem._2 }
    grouped.apply[HashMap[String, Int]](
      HashMap[String, Int](),
      (dict, x) =>
        if (dict.contains(x._1))
          dict + (x._1, dict(x._1) + 1)
        else
          dict + (x._1, 1)
    )
  }
```

Summary. In this section, we integrated multiple output producers to the staged fold API. This was done by implementing variants of the functions that delay the final application of fold by boxing elements into a type that preserves information about the multiple output separation. We also preserve the `FoldLeft` representation in the process.

These extra boxes unfortunately manifest in the generated code. In the next section we show how to eliminate this overhead.

5. Removing boxes

So far, we have successfully integrated `partition` and `groupBy` into the staged `FoldLeft` abstraction. Unfortunately this leads to the creation of boxes around elements. In this section, we discuss how to eliminate these boxes.

A first observation is that, inside the `FoldLeft` pipeline, we are free to choose any representation for our boxes, provided we can reconstruct the original representation at the end of the pipeline. In other words, we do not need to create instances of `Rep[Either[A, B]]` or `Rep[(K, V])` until the final application of `FoldLeft`. In particular, by using CPS-encoded versions of the boxes inside the pipeline, we can delay their construction, much like we delay the construction of lists. To illustrate this idea, we describe in this section a staged CPS encoding for the `Either` type, and show how to use it in the `partition` function.

5.1 EitherCPS

The CPS encoding for `Either` is given (unsurprisingly) by its functor representation:

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

`EitherCPS` *is* the function that abstracts over the eventual representation, `X`. It takes two functions that represent the left and right destructors yielding a value of type `X`.

```
trait EitherCPSOps extends Base with IfThenElse
  with BooleanOps {
  abstract class EitherCPS[A: Manifest, B: Manifest] { self =>
    def apply[X: Manifest](
      lf: Rep[A] => Rep[X],
      rf: Rep[B] => Rep[X]): Rep[X]

    def map[C: Manifest, D: Manifest](
      lmap: Rep[A] => Rep[C],
      rmap: Rep[B] => Rep[D]) = new EitherCPS[C, D] {
        def apply[X: Manifest](
          lf: Rep[C] => Rep[X],
          rf: Rep[D] => Rep[X]
        ) = self.apply(a => lf(lmap(a)), b => rf(rmap(b)))
      }
  }
  //Companion object
  object EitherCPS {
    def LeftCPS[A: Manifest, B: Manifest](a: Rep[A]) =
      new EitherCPS[A, B] {
        def apply[X: Manifest](
          lf: Rep[A] => Rep[X],
          rf: Rep[B] => Rep[X]) = lf(a)
      }

    def RightCPS[A: Manifest, B: Manifest](b: Rep[B]) =
      new EitherCPS[A, B] {
        def apply[X: Manifest](
          lf: Rep[A] => Rep[X],
          rf: Rep[B] => Rep[X]) = rf(b)
    }

    def conditional[A: Manifest, B: Manifest](
      cond: Rep[Boolean],
      thenp: => EitherCPS[A, B],
      elsep: => EitherCPS[A, B]
    ): EitherCPS[A, B] = {
      import lms.ZeroVal
      var l = ZeroVal[A]; var r = ZeroVal[B]
      var isLeft = true
      val lf = (a: Rep[A]) => { l = a; isLeft = true }
      val rf = (b: Rep[B]) => { r = b; isLeft = false }
      if (cond) thenp.apply[Unit](lf, rf)
      else elsep.apply[Unit](lf, rf)

      new EitherCPS[A, B] {
        def apply[X: Manifest](
            lf: Rep[A] => Rep[X],
            rf: Rep[B] => Rep[X]) =
          if (isLeft) lf(l) else rf(r)
      }
    }
  }
}
```

Figure 6. An implementation of staged EitherCPS

Having staged FoldLeft, staging EitherCPS is straightforward. Figure 6 gives an implementation for EitherCPS. In addition to map for functor application, and LeftCPS and RightCPS that create closures, we define a conditional combinator which handles conditional expressions. A naive implementation of conditional would simply wrap the conditional expression into a new instance of EitherCPS, applying its destructors in both branches. However, this duplicates the destructor code, and can quickly lead to code explosion. Instead, we bind the result of the respective branches to temporary variables before creating an instance of EitherCPS.

5.2 Tying the knot

Getting back to FoldLeft, we can now implement partition using EitherCPS. We face one final issue though. We may think that partition can be written as follows:

```
def partitionCPS(p: Rep[A] => Rep[Boolean]):
  FoldLeft[EitherCPS[A, A], S] = this map { elem =>
    if (p(elem)) LeftCPS[A, A](elem) else RightCPS[A, A](elem)
  }
```

However, FoldLeft expects a Rep type as its first argument. In this case, it expects a Rep[EitherCPS[A, A]] but we provide a plain EitherCPS[A, A]. At this point, having chosen LMS as our partial evaluation framework, we have no choice but to define an LMS intermediate representation for Rep[EitherCPS[A, A]]. Luckily, EitherCPS is already a code generator. So it suffices to add a simple IR wrapper around it which contains forwarder methods for every operator defined on EitherCPS. Figure 7 shows the implementation of this wrapper. We refer the interested reader to [12] for more details on the LMS IR.

6. Related Work

Fusion, or deforestation, has been studied extensively. One of the first known techniques is Wadler's algorithm for eliminating intermediate trees [16]. For list-like pipelines, there are three main algorithms: foldr/build fusion [5], which is based on implementing list operations as folds. Its dual, destroy/unfoldr fusion, fuses consumer functions such as zips, well [13]. Stream fusion [1, 2] converts list operations to operations on streams, and fuses both consumer and producer functions well. All three have been implemented using Haskell's rewrite rule system [7]. Although in this paper we only covered foldr/build fusion, we believe that our technique to the other two.

Fusion systems have also been studied theoretically. Meijer et al. [8] propose a theoretical framework for functional programs that are based on high-level recursive operations over algebras. The CPS-encoded datatypes (FoldLeft, EitherCPS) used in this paper are instances of such algebras. Hinze et al. provide a theoretical framework that unifies the above mentioned fusion algorithms [6]. Ghani et al. generalise foldr/build fusion to other inductive datatypes [4]. Although in this paper we only treat lists, sums and pairs,

```
trait EitherCPSOpsExp
    extends EitherCPSOps
    with BaseExp
    with IfThenElseExpOpt
    with BooleanOpsExpOpt
    with EqualExp {

  import EitherCPS._
  //The wrapper acts as a Rep[EitherCPS[A, B]]
  case class EitherWrapper[A, B](e: EitherCPS[A, B])
    extends Def[EitherCPS[A, B]]

  def mkLeft[A: Manifest, B: Manifest](a: Rep[A]) =
    EitherWrapper(LeftCPS[A, B](a))

  def mkRight[A: Manifest, B: Manifest](b: Rep[B]) =
    EitherWrapper(RightCPS[A, B](b))

  def eitherCPS_map[A: Manifest,
                    B: Manifest,
                    C: Manifest,
                    D: Manifest](
    e: Rep[EitherCPS[A, B]],
    lmap: Rep[A] => Rep[C],
    rmap: Rep[B] => Rep[D]
  ): Rep[EitherCPS[C, D]] = e match {
    case Def(EitherWrapper(sth)) =>
      EitherWrapper(sth map (lmap, rmap))
  }

  def either_apply[A: Manifest, B: Manifest, X: Manifest](
    e: Rep[EitherCPS[A, B]],
    lf: Rep[A] => Rep[X],
    rf: Rep[B] => Rep[X]
  ): Rep[X] = e match {
    case Def(EitherWrapper(sth)) => sth.apply(lf, rf)
  }

  def __ifThenElse[A: Manifest, B: Manifest](
    cond: Rep[Boolean],
    thenp: => Rep[EitherCPS[A, B]],
    elsep: => Rep[EitherCPS[A, B]]
  ): Rep[EitherCPS[A, B]] = (thenp, elsep) match {
    case (Def(EitherWrapper(t)), Def(EitherWrapper(e))) =>
      EitherWrapper(conditional(cond, t, e))
  }
}
```

Figure 7. EitherWrapper: LMS IR wrapper around Either-CPS

their work suggests that our technique can be extended to other inductive datatypes.

LMS also proposes its own fusion algorithm for indexed loops [11]. This algorithm performs both horizontal and vertical fusion on representations of loops and provides facilities for heterogeneous code generation. However, while the framework embraces the "fusion as a library" approach, it also relies heavily on LMS' compiler infrastructure. Our goal here was to avoid this kind of dependency, and implement a simple library based entirely on partial evaluation.

Partial evaluation and multi-stage programming have been used with great success to optimise programs. The general idea is to apply the first Futamura projection to turn interpreters into compilers [3]. The LMS framework enables us to compose code generators; we effectively operate in a generative programming language [9].

Svensson et al. use defunctionalization to unify push and pull arrays in an embedded DSL context [14]. Much like our approach, their representation effectively turns a CPS-encoded array into a code generator.

7. Conclusion and Future Work

We have shown how to implement fold-based fusion as a library. The key is to represent data-structures using their CPS-encodings. As a result, composition over these data structures turns into function composition. We then partially evaluate function composition to achieve vertical fusion.

The technique readily extends to multi-producers such as partitioning and grouping operations by introducing additional boxes. By CPS-encoding the box types, we are once again able to apply partial evaluation to eliminate intermediate data structures, and achieve horizontal fusion.

We used LMS as our staging/partial evaluation framework of choice: our implementation is available as an open-source project [1]. Our approach is, however, not tied to a particular framework. Indeed, any system capable of partially evaluating function composition is sufficient.

Our approach seems promising for other fusion techniques as well. In particular, we plan to extend our work to stream fusion, in hopes of making this powerful fusion technique available to a broader range of applications.

Acknowledgments

We thank members of the LAMP team at EPFL for many fruitful discussions and insightful suggestions. This research was sponsored by ERC under the DOPPLER grant (587327).

References

[1] D. Coutts. *Stream Fusion: Practical shortcut fusion for coinductive sequence types*. PhD thesis, University of Oxford, 2010.

[1] https://github.com/manojo/staged-fold-fusion

[2] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 315–326, New York, NY, USA, 2007. ACM.

[3] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[4] N. Ghani, P. Johann, T. Uustalu, and V. Vene. Monadic augment and generalised short cut fusion. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 294–305, New York, NY, USA, 2005. ACM.

[5] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM.

[6] R. Hinze, T. Harper, and D. W. H. James. Theory and practice of fusion. In *Proceedings of the 22Nd International Conference on Implementation and Application of Functional Languages*, IFL'10, pages 19–37, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] S. P. Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc, 2001.

[8] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[9] T. Rompf, K. Brown, H. Lee, A. Sujeeth, M. Jonnalagedda, N. Amin, Y. Klonatos, M. Dashti, C. Koch, and K. Olukotun. Go meta! for a fundamental shift towards generative programming and dsls in performance critical systems. In *Proceedings of the Inaugural Summit on Advances in Programming Languages*, SNAPL 2015, 2015.

[10] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 127–136, New York, NY, USA, October 10–13 2010. ACM.

[11] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 497–510, New York, NY, USA, 2013. ACM.

[12] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented dsls. In *Proceedings IFIP Working Conference on Domain-Specific Languages, DSL 2011, Bordeaux, France, 6-8th September 2011.*, pages 93–117, 2011.

[13] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 124–132, New York, NY, USA, 2002. ACM.

[14] B. J. Svensson and J. Svenningsson. Defunctionalizing push arrays. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '14, pages 43–52, New York, NY, USA, 2014. ACM.

[15] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[16] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, Jan. 1988.