Polarized Data Parallel Data Flow

Ben Lippmeier[α] Fil Mackay[β] Amos Robinson[γ]

[α,β]Vertigo Technology (Australia) [α,γ]UNSW (Australia) [γ]Ambiata (Australia)

{benl,fil}@vergo.co {benl,amosr}@cse.unsw.edu.au amos.robinson@ambiata.com

Abstract

We present an approach to writing fused data parallel data flow programs where the library API guarantees that the client programs run in constant space. Our constant space guarantee is achieved by observing that binary stream operators can be provided in several *polarity versions*. Each polarity version uses a different combination of stream sources and sinks, and some versions allow constant space execution while others do not. Our approach is embodied in the Repa Flow Haskell library, which we are currently using for production workloads at Vertigo.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers; Optimization

Keywords Arrays; Fusion; Haskell

1. Introdution

The functional language ecosystem is blessed with a multitude of libraries for writing streaming data flow programs. Stand out examples for Haskell include Iteratee [8], Enumerator [14], Conduit [17] and Pipes [5]. For Scala we have Scalaz-Streams [3] and Akka [9].

Libraries like Iteratee and Enumerator guarantee that the client programs run in constant space, and are commonly used to deal with data sets that do not fit in main memory. However, the same libraries do not provide a notion of *parallelism* to help deal with the implied amount of data. They also lack support for branching data flows where produced streams are consumed by several consumers without the programmer needing to hand fuse the consumers. We provide several techniques to increase the scope of such stream processing libraries:

- Our parallel data flows consist of a bundle of streams, where each stream can be processed in a separate thread. (§2)
- Our API uses polarized flow endpoints (`Sources` and `Sinks`) to ensure that programs run in constant space, even when produced flows are consumed by several consumers. (§2.3)
- We show how to design the core API in a generic fashion so that chunk-at-a-time operators can interoperate smoothly with element-at-a-time operators. (§3)

FHPC'16, September 22, 2016, Nara, Japan
ACM. 978-1-4503-4433-3/16/09...
http://dx.doi.org/10.1145/2975991.2975999

- We use continuation passing style (CPS) to provoke the Glasgow Haskell Compiler into applying stream fusion across chunks processed by independent flow operators. (§3.1)

Our target applications concern *medium data*, meaning data that is large enough that it does not fit in the main memory of a normal desktop machine, but not so large that we require a cluster of multiple physical machines. For lesser data one could simply load it into main memory and use an in-memory array library. For greater data one needs to turn to a distributed system such as Hadoop [16] or Spark [19] and deal with the unreliable network and lack of shared memory. Repa Flow targets the sweet middle ground.

2. Streams and Flows

A *stream* is an array of elements where the indexing dimension is time. As each element is read from the stream it is available only in that moment, and if the consumer wants to re-use the element at a later time it must save it itself. A *flow* is a bundle of related streams, where each stream carries data from a single partition of a larger data set — we might create a flow consisting of 8 streams where each one carries data from a 1GB partition of a 8GB data set. We manipulate flow endpoints rather than the flows themselves, using the following data types:

```
data Sources i m e
   = Sources
   { arity :: i
   , pull  :: i -> (e -> m ()) -> m () -> m () }

data Sinks   i m e
   = Sinks
   { arity :: i
   , push  :: i -> e -> m ()
   , eject :: i -> m () }
```

Type `Sources i m e` classifies flow sources which produce elements of type `e`, using some monad `m`, where the streams in the bundle are indexed by values of type `i`. The index type `i` is typically `Int` for a bundle of many streams, and `()` for a bundle of a single stream. Likewise `Sinks i m e` classifies flow sinks which consume elements of type `e`.

In the `Sources` type, field `arity` stores the number of individual streams in the bundle. To receive data from a flow producer we apply the function in the `pull` field, passing the index of type `i` for the desired stream, an *eat* function of type `(e -> m ())` to consume an element if one is available, and an *eject* computation of type `(m ())` which will be invoked when no more elements will ever be available for that stream. The pull function will then perform its `(m ())` computation, for example reading a file, before calling our eat or eject, depending on whether data is available.

In the `Sinks` type, field `arity` stores the number of individual streams as before. To send data to a flow consumer we apply the `push` function, passing the stream index of type `i`, and the element of type `e`. The `push` function will perform its `(m ())` computation to consume the provided element. If no more data is available we instead call the `eject` function, passing the stream index of type `i`, and this function will perform an `(m ())` computation to shut down the sink — possibly closing files or disconnecting sockets.

Consuming data from a `Sources` and producing data to a `Sinks` is synchronous, meaning that the computation will block until an element is produced or no more elements are available (when consuming); or an element is consumed or the endpoint is shut down (when producing). The `eject` functions used in both `Sources` and `Sinks` are associated with a single stream only, so if our flow consists of 8 streams attached to 8 separate files then ejecting a single stream will close a single file.

2.1 Sourcing, Sinking and Draining

Figure 1 gives the definitions of `sourceFs`, `sinkFs` which create flow sources and sinks based on a list of files (`Fs` = files), as well as `drainP` which pulls data from a flow source and pushes it to a sink in `P`-arallel. We elide type class constraints to save space.

Given the definition of the `Sources` and `Sinks` types, writing `sourceFs` and `sinkFs` is straightforward. In `sourceFs` we first open all the provided files, yielding file handles for each one, and the `pull` function for each stream source reads data from the corresponding file handle. When we reach the end of a file we eject the corresponding stream. In `sinkFs`, the `push` function writes the provided element to the corresponding file, and `eject` closes it.

The `drainP` function takes a bundle of stream `Sources`, a bundle of stream `Sinks`, and drains all the data from each source into the corresponding sink. Importantly, `drainP` forks a separate thread to drain each stream, making the system data parallel. We use Haskell MVars for communication between threads. After forking the workers, the main thread waits until they are all finished. Now that we have the `drainP` function, we can write the "hello world" of data parallel data flow programming: copy a partitioned data set from one set of files to another:

```
copySetP :: [FilePath] -> [FilePath] -> IO ()
copySetP srcs dsts
 = do  ss <- sourceFs srcs
       sk <- sinkFs   dsts
       drainP ss sk
```

2.2 Stateful Streams, Branching and Linearity

Suppose we wish to copy our files while counting the number of characters copied. To count characters we can `map` each to the constant integer one, then perform a `fold` to add up all the integers. To avoid reading the input data from the file system twice we use a counting process that does not force the evaluation of this input data itself. For this reason we use versions of `map` and `fold` that produce new `Sinks` rather than `Sources`. Their types are as follows:

```
map_o  :: (a -> b) -> Sinks i m b -> Sinks i m a

fold_o :: Ord i => i -> (a -> a -> a) -> a
                -> IO (Sinks i IO a, IORef a)
```

The `map_o` operator transforms a sink of `b` things into a sink of `a` things. The `fold_o` operator produces a sink for `a` things, and an `IORef`. The value in the `IORef` is initialised to the given starting value of type `a`, which is combined with elements pushed to the sink using the worker function of type `(a -> a -> a)`. The first argument of type `i` is the arity of the result sinks.

```
sourceFs :: [FilePath] -> IO (Sources Int IO Char)
sourceFs names
 = do hs <- mapM (\n -> openFile n ReadMode) names
      let pulls i ieat ieject
          = do let h = hs !! i
               eof <- hIsEOF h
               if eof then hClose   h >> ieject
                      else hGetChar h >>= ieat
      return (Sources (length names) pulls)

sinkFs  :: [FilePath] -> IO (Sinks Int IO Char)
sinkFs names
 = do hs <- mapM (\n -> openFile n WriteMode) names
      let pushs  i e = hPutChar (hs !! i) e
      let ejects i   = hClose   (hs !! i)
      return (Sinks (length names) pushs ejects)

drainP :: Sources i IO a -> Sinks i IO a -> IO ()
drainP (Sources i1 ipull) (Sinks i2 opush oeject)
 = do mvs <- mapM makeDrainer [0 .. min i1 i2]
      mapM_ readMVar mvs
 where
  makeDrainer i = do
    mv <- newEmptyMVar
    forkFinally (newIORef True >>= drainStream i)
                (\_ -> putMVar mv ())
    return mv
  drainStream i loop =
    let eats v = opush i v
        ejects = oeject i >> writeIORef loop False
    in  while (readIORef loop) (ipull i eats ejects)
```

Figure 1. Sourcing, Sinking and Draining

Besides `map_o` and `fold_o` we also need an operator to branch the flow so that the same data can be passed to our counting operators as well as written back to the file system.

The branching operator we use is as follows:

```
dup_ooo :: (Ord i, Monad m)
       => Sinks i m a -> Sinks i m a -> Sinks i m a
dup_ooo (Sinks n1 push1 eject1)
        (Sinks n2 push2 eject2)
 = let pushs  i x = push1 i x >> push2 i x
       ejects i   = eject1 i  >> eject2 i
   in  Sinks (min n1 n2) pushs ejects
```

This operator takes two argument sinks and creates a new one. When we push an element to the new sink it will push that element to the two argument sinks. Likewise, when we eject a stream in the new sink it will eject the corresponding stream in the two argument sinks. We can use this new combinator to write a working `copyCountP` function:

```
copyCountP :: [FilePath] -> [FilePath] -> IO Int
copyCountP srcs dsts
 = do  ss      <- sourceFs srcs
       sk1     <- sinkFs   dsts
       (sk2,r) <- fold_o  (arity ss) (+) 0
       drainP ss (dup_oo sk1 (map_o (const 1) sk2))
       readIORef r
```

This function runs in constant space, using a single pass over the input data, which is the behaviour we wanted. Note that in the definition there is only a single occurrence of each of the variables bound to sources and sinks: `ss`, `sk1`, `sk2`. Each source and sink is used linearly. Our program expresses a data flow graph where the functions `sourceFs`, `sinkFs`, `fold_o`, `map_o` and `dup_ooo` create nodes, and the use-def relation of variables defines the edges.

Linearity is an important point. Suppose we instead tried to compute our result in two separate stages, one to copy the files to their new locations and one to compute the count:

```
badCopyCount :: [FilePath] -> [FilePath] -> IO Int
badCopyCount srcs dsts
 = do  ss      <- sourceFs srcs
       sk1     <- sinksFs  dsts
       drainP ss sk1
       (sk2,r) <- fold_o (+) 0 (arity ss)
       drainP ss (map_o (const 1) sk2)
       readIORef r
```

This cannot work. The `Sinks` endpoint created by `sourceFs` is a stateful object – it represents the current position in each of the source files being read. After we have applied the first `drainP`, we have already finished reading through all the source files, so draining the associated flow again does not yield more data. In general an object of type `Sources` is an abstract producer of data, and it may not even be possible to rewind it to a previous state — suppose it was connected to a stream of sensor readings. Alas the Haskell type system does not check linearity so we rely on the programmer to enforce it manually.

2.3 Polarity and Buffering

Our `dup_ooo` operator from the previous section branches a flow by taking two existing sinks and producing a new one. The `_ooo` suffix stands for "output, output, output", referring to the three sinks. It turns out that the converse `dup_iii` operator is not implementable without requiring unbounded buffering. Such an operator would have the following type:

```
 dup_iii :: (Ord i, Monad m)
         =>  Sources i m a
         -> (Sources i m a, Source i m a)
```

Consider how this would work. The `dup_iii` operator takes an argument source and produces two result sources. Now suppose we pull data from the left result source. The operator would need to pull from its argument source to retrieve the data, then when we pull from the right result source we want this same data. The problem is that there is nothing stopping us from pulling the entire stream via the left result source before pulling any elements from the right result source, so `dup_iii` would need to introduce an unbounded buffer to store all elements in the interim.

Interestingly, although `dup_iii` cannot work without an unbounded buffer, a hybrid operator `dup_ioi` can. This operator has the following definition:

```
dup_ioi :: (Ord i, Monad m)
        => Sources i m a -> Sinks i m a
        -> Sources i m a
dup_ioi (Sources n1 pull1) (Sinks n2 push2 eject2)
 = let pull3 i eat3 eject3
        = pull1 i eat1 eject1
        where eat1 x = eat3 x >> push2  i x
              eject1 = eject3 >> eject2 i
   in  Sources (min n1 n2) pull3
```

The `dup_ioi` operator takes an argument source, an argument sink, and returns a result source. When we pull data from the result the operator pulls from its argument source and then *pushes* the same data to the argument sink. Similarly to `dup_ooo`, we can use `dup_ioi` to introduce a branch into the data flow graph *without* requiring unbounded buffering of the input flow. This fact was also noticed in [2].

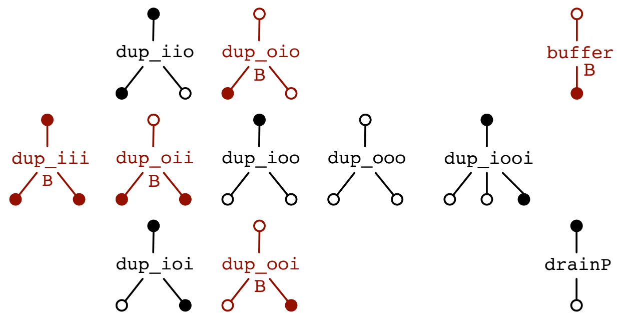


Figure 2. Possible polarities for the flow duplication operator

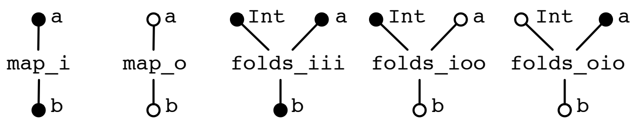


Figure 3. Polarities for `map` and `folds`

We can extend `dup_iooi` to work with any number of argument sinks, for example:

```
 dup_iooi
  :: (Ord i, Monad m)
  => Sources i m a -> Sinks i m a -> Sinks i m a
  -> Sources i m a
```

With `dup_ioi` when we pull from the result source the operator pulls from its argument source and pushes the same data to its argument sinks.

Figure 2 shows the possible *polarity versions* for the flow duplication operator. Sources are indicated with a • and sinks with a ◦. We use the mnemonic that the filled • can always produce data (being a source) while the empty ◦ can always accept data (being a sink). In the figure the versions marked with a `B` (in red) would require unbounded `B`-uffering, while the black ones require no buffering.

In the right of Figure 2 we have also included the corresponding diagrams for the `drainP` operator from Figure 1. The `drainP` operator requires no buffering because all data pulled from the source is immediately pushed to the sink. On the other hand, if we invert the polarities of `drainP` we arrive at the natural assignment for a primitive buffering operator. Any operator that accepts data via a single argument sink and produces the same data via a result source *must* introduce a buffer, as there is no guarantee that new elements are pushed to the sink at the same rate they are pulled from the source. Our Repa Flow library guarantees that programs written with it run in constant space by only providing operators in the polarity versions that do so.

2.4 Mapping

We mentioned the `map_o` operator back in §2.2. Here is its type again, along with the matching `map_i` with inverse polarity:

```
map_o :: (a -> b) -> Sinks   i m b -> Sinks   i m a
map_i :: (a -> b) -> Sources i m a -> Sources i m b
```

The first form is a sink transformer, taking a sink of elements of type `b` and returning a sink of elements of type `a`. The second is a source transformer, taking a source of elements of type `a` and returning a source of elements of type `b`. The polarity diagram for both forms is given in Figure 3. In both cases the input elements have type a and the output elements have type b. The definition of map_i is as follows:

```
map_i :: (a -> b) -> Sources i m a -> Sources i m b
map_i f (Sources n pullsA)
 = Sources n pullsB
 where  pullsB i eatB ejectB
         = pullsA i eatA ejectA
         where  eatA v = eatB (f v)
                ejectA = ejectB
```

Note that both map_i and map_o are simply flow transformers, and are neither inherently parallel or sequential. Repa Flow provides data parallelism, and this parallelism is introduced by the singular drainP function. In this respect drainP is similar to the computeP function over delayed arrays from our original Repa library [11], except that parallelism is introduced on a per-stream level rather than a per-element level. We will return to the definition of map_i when we discuss chunking in §3.

2.5 Segmented Folding

Figure 3 includes polarity diagrams for three forms of segmented fold. The first one, folds_iii has the following type:

```
folds_iii :: (Ord i, Monad m) => (b -> a -> b) -> b
          -> Sources i m Int  -> Sources i m a
          -> Sources i m b
```

The folds_iii operator takes a flow of segment lengths, a flow of elements, and uses the provided combining function and neutral value to fold segments from each stream in the flow. For example, suppose we have the following flows, writing the streams of elements in each flow using nested list syntax:

```
folds_iii (+) 0
   [ [3      2   1] [2   2  ] [4        ] ]
   [ [1 2 3 1 1 5] [3 3 4 4] [4 3 2 1] ]
 = [ [6      2   5] [6   8  ] [10       ] ]
```

For the first stream the segment lengths are [3 2 1] and the elements are [1 2 3 1 1 5], we sum up the first three elements, then the next two, then the next one, yielding [6 2 5] for that stream. When drainP from Figure 1 is applied to the result source, each of the individual streams in the flow is folded in parallel.

With folds_iii we assume that both the segment lengths and elements are available as sources. When this is true, evaluation of folds_iii requires no buffering. When we pull a fold result from the result source, the operator pulls the segment length from its argument source, then the corresponding elements from the element source, folding them in the process.

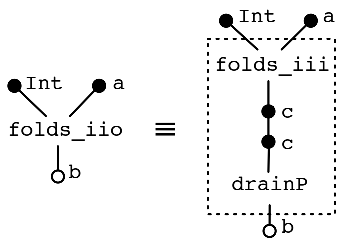
As per Figure 3 we can assign polarities to folds in two other ways that allow the operator to execute without buffering.

With folds_ioo we push elements to the element sink (on the right). As needed, the operator pulls segment lengths from the segment length source (on the left), which instruct it how many of the consecutive elements to fold. As each segment is completed it pushes the result to the result sink (on the bottom).

With folds_oio we push segment lengths to the segment length sink (on the left). As needed, the combinator pulls the corresponding number of elements from the element source (on the right) and folds them. As each segment is completed it pushes the result to the result sink (on the bottom).

One might wonder if folds_iii, folds_ioo and folds_oio are the only versions that can execute without buffering. There are a total of 8 polarity versions for a 3-leg operator such as folds.

Case analysis reveals that the others require unbounded buffering, except for the following special version:



The folds_iio version does not require buffering because all segment lengths and elements are available as sources, and the result of folding each segment can be pushed directly to the result sink. However, this version is not primitive as it can be expressed as the composition of folds_iii and drainP as shown above. We refer to a polarity version that can be expressed as a composition with drainP as an *active version*, because its form implies computation rather than being an *inactive* transformation on sources on sinks. Note that dup_ioo from Figure 2 is also active.

In our Repa Flow library we provide only polarity versions that execute without buffering, and the only active versions are drainP and drainS (sequential drain). This restriction ensures that it is easy for the programmer to reason about when computation happens, as well as having the programs execute in constant space.

2.6 Stream Projection, Funneling, and Fattening

So far the operators we have discussed have all performed the same computation on each stream in the flow. Here are four basic operators to convert between flows consisting of several streams and *singleton* flows, containing only one stream. The endpoints for singleton flows have the index type set to () which indicates there is only one stream in the flow.

```
project_i :: i -> Sources i m a -> Sources () m a
project_o :: i -> Sinks   i m a -> Sinks   () m a

funnel_i :: Sources i IO a ->  IO (Sources () IO a)
funnel_o :: i -> Sinks () IO a -> IO (Sinks i IO a)
```

The project operators each take a stream index, an endpoint for a flow of several streams, and return an endpoint for a singleton flow containing only the specified stream. The project_i operator takes a flow source of several streams, and returns a flow source that selects only the specified stream. The project_o operator takes a flow sink for several streams and returns a sink that discards data in all streams except the specified one.

The funnel operators each take an endpoint for a flow of several streams, and return a singleton flow containing *all* data in the argument flow. The funnel_o opeartor also takes the desired arity of the result Sinks. These operators expose a duality in the inversion of control associated with a stream combinator library:

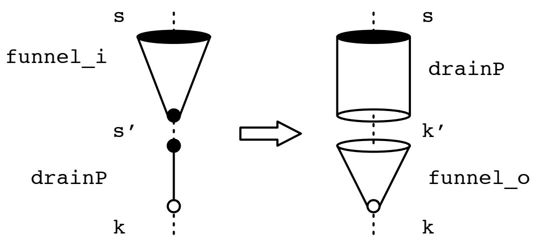
With funnel_i the order in which the argument streams are processed is under the control of the *operator* and is *deterministic* when viewed by the consumer of the result source. In our implementation the default order is to send data from each argument stream from lowest index to highest. Other orders are possible, such as a round-robin process that produces a single element from each non-empty stream in turn.

In contrast, with funnel_o the order in which argument streams are processed is controlled by the *context* and is *non-deterministic* when viewed by the consumer attached to the argument sink. Recall that in the implementation of drainP from Figure 1 we forked a thread to evaluate each source stream. As each of these threads pushes to its corresponding sink concurrently, if these sinks are then funneled then the order in which elements appear in the output may vary from run to run. As the sink may be attached to a shared resource such as a file, this is also the place in the library where we must introduce locks to manage contention. Other operators such as `map` and `folds` are lock free, as although they are applied to flow endpoints that conceptually carry several streams at once, at runtime there is no communication between the threads that evaluate each of these streams.

The fact that `funnel_o` is more complex to implement than `funnel_i` stems from the "central dogma" of data flow programming: information flows from inputs to outputs. If it also flowed the other way then `funnel_i` would also need locks. The central dogma suggests a natural program transformation, which we name *drain fattening*:

```
   (funnel_i            s >>= λs'. drainP s' k)
=> (funnel_o (arity s) k >>= λk'. drainP s  k')
```

This is better expressed as a picture:



Drain fattening is valid provided the consumer of the final result source `k` performs a commutative reduction, or is otherwise insensitive to the order in which elements arrive. By exchanging the order of `drainP` and `funnel` we allow parallel evaluation of multiple streams in the flow, at the expense of introducing non-determinism in the order in which elements are pushed to the sink. Drain fattening expresses the change in computation structure that arises when moving from a process that performs sequential reduction of a data set, to one that performs parallel tree reduction. However, in our case the change in computation structure is separated from the need to actually perform a reduction.

Finally, although the syntactic transform requires `funnel_i` to be close to `drainP`, the fact that our operators come in a variety of polarity versions allows us to pull many through a `drainP` to bring their arguments closer, for example:

```
drainP (map_i f s) k  =  drainP s (map_o f k)
```

3. Chunked Streams

The flows we have discussed so far have processed elements of a generic type `e`. Although we can instantiate our operators at atomic types such as `Char` and `Float`, in practice to gain reasonable runtime performance we must amortize the cost of control flow by processing a chunk containing several elements at a time. We instantiate the generic `Sources` and `Sinks` types to produce chunked versions `CSources` and `CSinks`:

```
type CSources i m e = Sources i m (Vector e)
type CSinks   i m e = Sinks   i m (Vector e)
```

In our Haskell implementation we use the unboxed `Vector` type from the standard `vector` library to represent chunks. For operators that do not inspect the actual elements, such as `drainP` and `funnel_i`, their chunked versions are simply aliases for the generic versions, but with more specific types so that the API documentation is easier to read. Other operators such as the `folds` family need separate implementations because their argument functions (of type `(a -> b -> a)`) work on single elements rather than chunks.

3.1 Intra-chunk Fusion

Fusion of operators on chunked streams arises naturally from the array fusion system already implemented in the `vector` library, which processes the chunks. This happy situation is due to the fact that our generic flow operators are written in continuation passing style. For example, here is the definition of the map function for sources of chunked flows. We use the existing `map_i` operator on generic flows, and the `umap` operator on unboxed vectors. We suppress type class constraints to save space.

```
cmap_i :: (a -> b)
       -> CSource i m e -> CSource i m e
cmap_i f ss = map_i (umap f) ss
```

Now, suppose we map a per-element function `g` over a flow, then map another per-element function `f` to this result. Both `f` and `g` apply to all elements of all streams in each flow.

```
cmap_i f (cmap_i g ss)
```

We expand the `Sources` value `ss`, naming the arity component `n` and the pull function `pulls`.

```
cmap_i f (cmap_i g (Sources n pulls))
```

Inlining the definition of `cmap_i` above gives:

```
=> map_i (umap f) (map_i (umap g) (Sources n pulls))
```

Inlining the definition of `map_i` and simplifying then yields:

```
=> Sources n (λi eat eject.
     pulls i (λv. eat (umap f (umap g v))) eject)
```

The two instances of `umap` are now syntactically adjacent, which allows the fusion system in the vector library to fire:

```
=> Sources n (λi eat eject.
     pulls i (λv. eat (umap (f ∘ g) v)) eject)
```

Suppose we pull some data from this flow source. We apply the contained function to the index `i` of the stream we are interested in, and provide our own `eat` and `eject` functions to either to consume a chunk or indicate that there is no more data available in stream `i`. The flow source would then apply the `pulls` function from *its* own parent source to the inner continuation. If a chunk from the parent source is available, then the `umap` function will then apply the fused `f` and `g` to each element, before passing the result to the original `eat` function that we provided.

3.2 Leftovers

When flows carry chunks instead of single elements it becomes naturally harder to write operators that consume only a few elements at a time, rather than a whole chunk at a time. Consider the `head_i` operator which splits the first element from a specified argument stream, as well as producing a flow of the leftover elements:

```
head_i :: i -> Sources i m a -> (a, Sources i m a)
```

Libraries like `conduit` [17] manage leftovers by extending the representation of stream sources with a special constructor that carries an array of leftover elements from a previous chunk, as well as the continuation to pull more chunks from the source. For Repa Flow we avoid adding more constructors to our `Source` and `Sink` data types, as moving away from the simple continuation passing style of `pull`, `eat` and `eject` makes it harder to perform the program simplifications that enable intra-chunk fusion. Instead, operators such as `head` produce a new flow source where the embedded `pull` function first produces a chunk containing leftover elements before pulling more chunks from its own source. The chunk of leftover elements is stored in the closure of the pull function itself, rather than being reified into the representation of the `Source` data type. We rely on the linearity convention to ensure the argument source is not reused, as applying `head_i` to the same source would yield the first element in the *next* chunk, rather than the next element after the one that was previously returned.

4. Related Work

Our work is embodied in Repa Flow, which is available on Hackage. Repa Flow is a new layer on top of the existing Repa library for delayed arrays [11], and performs fusion via the GHC simplifier rather than using a custom program transformation based on series expressions [12] as in our prior work.

Our system ensures that programs run in constant space, without requiring buffering or backpressure as in Akka [9] and Heron [10]. Our stream programs are fused into nested loops that read the input data from the source files, process it, and write the results immediately without blocking. The only intermediate space required is for aggregators — for example if we had a stream of text and were counting the number of occurrences of each word we would need a map of words to the number of occurrences.

We use stream processing to deal with large data sets that do not fit in memory. Real time streaming applications, such as to process click streams generated from websites, are the domain of synchronous data flow languages such as Lucy-n [13], and reactive stream processing systems such as Heron [10] and S4 [15]. Lucy uses a clock analysis to determine where (finite) buffers must be introduced into the data flow graph. Heron and S4 use fixed size buffers with runtime back-pressure to match differing rates of production and consumption.

The main difference between Repa Flow and Iteratee based Haskell libraries [5, 8, 14, 17] is that Repa Flow uses the separate `Sources` and `Sinks` types to express the *endpoints* of flows, whereas an Iteratee is better thought of as a *computation* as it is given a monadic interface. The advantage of the `Iteratee` approach is that pleasing algebraic identities arise between iteratee computations. The disadvantage is that consuming data from two separate sources is awkward because each source is represented by its own monadic computation, and multiple computations must be layered using monad transformers. Repa Flow lacks the convenience of a uniform monadic interface, though writing programs that deal with many sources and sinks is straightforward by design.

The idea that parallelism can be introduced into a data flow graph via a single operator is well known in the databases community. The Volcano [6] parallel database inserts an `exchange` operator into its query plans, which forks a child thread for the producer of some data, leaving the master thread as the consumer. The implementation of `exchange` also introduces buffering and uses back pressure to handle mismatch between rates of production and consumption. In Repa Flow we use `drainP` to introduce parallelism, and `drainP` itself introduces no extra buffering. In Volcano and other database systems, communication between operators is performed with a uniform `open`, `next`, `close` interface, similar to a streaming file API. In Repa Flow the API between operators consists of the `Sources` and `Sinks` type, where the next element in a given stream can be uniformly acquired via the `pull` function.

In itself the duality between source and sink, push and pull, is folklore, and has previously been used for code generation in array processing languages [4, 18] and XML processing pipelines [7]. More recently, Bernardy and Svenningsson describe a library [2] that defines streams with sources and sinks, where each is defined as if it were the logical negation of the other. They also define co-sources and co-sinks, where a co-source is a sink that accepts element consumers and a co-sink is a source that produces element consumers. In related work Bernardy *et al* describe a core calculus [1] based on Linear Logic which guarantees fusion does not increase the cost of program execution. The system is based fundamentally around linear logic rather than lambda calculus, with evaluation being driven by cut elimiation rather than function application. They describe a compiler targeting C and encouraging benchmark results.

References

[1] Jean-Philippe Bernardy, Víctor López Juan, and Josef Svenningsson. Composable efficient array computations using linear types. Unpublished Draft, 2016.

[2] Jean-Philippe Bernardy and Josef Svenningsson. On the duality of streams. how can linear types help to solve the lazy IO problem? In *IFL: Implementation and Application of Functional Languages*, 2015.

[3] Paul Chiusano and Pavel Chlupacek et al. The scalaz-streams library. `http://github.com/functional-streams-for-scala/fs2`.

[4] Koen Claessen, Mary Sheeran, and Joel Svensson. Expressive array constructs in an embedded GPU kernel programming language. In *DAMP: Declarative Aspects of Multicore Programming*, 2012.

[5] Gabriel Gonzalez. The pipes Haskell package. `http://hackage.haskell.org/package/pipes`.

[6] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge Data Engineering*, 6(1), 1994.

[7] Michael Kay. You Pull, I'll Push: on the Polarity of Pipelines. In *Balisage: The Markup Conference*, 2009.

[8] Oleg Kiselyov. Iteratees. In *FLOPS: Functional and Logic Programming*, 2012.

[9] Viktor Klang and Patrik Nordwall et al. The Akka project. `http://github.com/akka/akka`.

[10] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream processing at scale. In *SIGMOD: International Conference on Management of Data*, 2015.

[11] Ben Lippmeier, Manuel M. T. Chakravarty, Gabriele Keller, and Simon L. Peyton Jones. Guiding parallel array fusion with indexed types. In *Haskell Symposium*, 2012.

[12] Ben Lippmeier, Manuel M. T. Chakravarty, Gabriele Keller, and Amos Robinson. Data flow fusion with series expressions in Haskell. In *Haskell Symposium*, 2013.

[13] Louis Mandel, Florence Plateau, and Marc Pouzet. Lucy-n: a n-synchronous extension of Lustre. In *Mathematics of Program Construction*, 2010.

[14] John Millikin and Mikhail Vorozhtsov. The enumerator Haskell package. `http://hackage.haskell.org/package/enumerator`.

[15] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: distributed stream computing platform. In *ICDMW: International Conference on Data Mining*, 2010.

[16] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST: Mass Storage Systems and Technologies*, 2010.

[17] Michael Snoyman. The conduit Haskell package. `http://hackage.haskell.org/package/conduit`.

[18] Bo Joel Svensson and Josef Svenningsson. Defunctionalizing push arrays. In *FHPC: Functional High-Performance Computing*, 2014.

[19] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI: Networked Systems Design and Implementation*, 2012.