The stuff that streams are made of

Streaming models for concurrent execution of multiple queries

Amos Robinson

A thesis in fulfilment of the requirements for the degree of
Doctor of Philosophy

School of Computer Science and Engineering
Faculty of Engineering
University of New South Wales
October 17, 2018

CONTENTS

ABSTRACT

To learn from a large dataset, we generally want to perform lots of queries. If we perform each query separately, we may spend more time reading and re-reading the same dataset than we spend computing the answer. Instead of performing each query separately, we would like to amortise the cost of reading the data by performing multiple queries at the same time.

Two streaming models for executing multiple queries concurrently are push streams and Kahn process networks. Push streams can be used to execute multiple queries concurrently, but push streams can be unwieldy to use as queries must be constructed "back-to-front". We introduce a query language called Icicle, which allows programmers to write and reason about queries using a more familiar array-based semantics, while retaining the execution strategy of push streams. The type system of Icicle guarantees that well-typed query programs have the same semantics whether they are executed as array programs or as stream programs, and that all queries over the same input data can be executed together.

However, push streams do not support computations with multiple inputs except for non-deterministically merging two streams. Kahn process networks support multiple inputs and multiple queries, but require dynamic scheduling and inter-process communication, both of which can introduce significant overhead. We introduce a method for taking multiple processes in a Kahn process network and fusing them together into a single process. The fused process communicates through local variables rather than costly communication channels. This fusion method generalises previous work on stream fusion and demonstrates the connection between fusion and the synchronised product operator, which is generally used in the context of verification and model checking, rather than as an optimisation.

Some queries must be executed in multiple passes, as they need to read the input data multiple times, or may produce intermediate outputs which are then read back in. For such queries, there are usually many different ways to schedule the work among the separate passes. Prior work demonstrated how integer linear programming (ILP) can be used to find optimal schedules for imperative array programs. However, these approaches can only handle operations that preserve the size of the array, missing out on some optimisation opportunities. We introduce a clustering algorithm for scheduling queries written using array combinators, and extend prior work to support size-changing operations.

ACKNOWLEDGEMENTS

Thanks to my supervisors: Ben, Gabi, and Manuel. All three have moved on from UNSW, but they nonetheless continue to support me greatly.

Thanks to my partner Laurel, to my dear friend James, and to the rest of my family.

PUBLICATIONS

During the course of my research, I coauthored the following publications:

- Lippmeier, B., Chakravarty, M. M. T., Keller, G., and Robinson, A. (2013). Data flow fusion with series expressions in Haskell. In *Proceedings of the 2013 Haskell symposium*. In Submission
- Robinson, A., Lippmeier, B., and Keller, G. (2014). Fusing filters with integer linear programming. In *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*, pages 53–62. ACM
- Robinson, A. and Lippmeier, B. (2016). Icicle: write once, run once. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing*, pages 2–8. ACM
- Lippmeier, B., Mackay, F., and Robinson, A. (2016). Polarized data parallel data flow. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing*
- Robinson, A. and Lippmeier, B. (2017). Machine fusion: merging merges, more or less. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, pages 139–150. ACM

CHAPTER 1

INTRODUCTION

To learn interesting things from a large dataset, we generally want to perform lots of queries. When each query is simple and our data is big, we might spend more time reading the data than we spend computing the answer. Instead of performing each query separately, we would like to amortise the cost of reading the data by performing multiple queries at the same time.

When querying datasets that do not fit in memory or disk, it can be hard to ensure that our query program's internal state will fit in memory. One way to transform large datasets in constant memory is to write the query as a *streaming program* (Chapter 2) by composing stream transformers together. Composing stream transformers naively can introduce some performance overhead, which is then usually removed by *fusing* multiple transformers into a single transformer.

This thesis describes low-overhead streaming models for executing multiple queries at a time. We focus on two streaming models: push streams (Section 2.3), and Kahn process networks (Section 2.5, Chapter 4).

Push streams can be used to execute multiple queries at a time, but queries using push streams can be unwieldy to write as they must be constructed "back-to-front". In Chapter 3 we introduce a query language called Icicle, which allows programmers to write and reason about queries using a more familiar array-based semantics, while retaining the execution strategy of push streams. The type system of Icicle guarantees that well-typed query programs have the same semantics whether they are executed as array programs or as stream programs, and that all queries over the same input can be executed together in a single pass. Icicle has been running in production at Ambiata, a machine-learning company, for over two years. For one particular client, we receive thirty gigabtyes of uncompressed new data every day. Each record in the data represents an action by a particular end-user, and we use Icicle to execute a set of around four thousand prepared queries for each end-user. Each end-user's query results are given as inputs to a statistical model that aims to predict the end-user's future behaviour. The compressed input data is stored indefinitely and currently requires five terabytes of storage. When creating a new statistical model, Icicle is used to provide training data by executing a new set of queries over the entire historical data.

However, push streams, and by extension Icicle, do not support streaming computations with multiple inputs except for non-deterministically merging two streams. In some circumstances, this non-deterministic merge can be used to append streams — as we shall see in Section 2.3. As an alternative to push streams, Kahn process networks (Kahn et al., 1976) support both multiple inputs and multiple queries, but require dynamic scheduling and inter-process communication, both of which can introduce significant runtime overhead. In Section 4.3 we describe *process fusion*, a method for taking multiple processes in a Kahn process network and fusing them together into a single process. The fused process communicates through local variables rather than costly communication channels, and executes faster than the original Kahn process network in our benchmarks (Chapter 5). This fusion method generalises previous work on stream fusion (Section 6.1) and demonstrates the connection between fusion and the synchronised product operator (Section 6.4), which is generally used in the context of verifying properties of concurrent processes and model checking, rather than as an optimisation.

When the input dataset is small enough to fit in memory, we can write each query as an *array program* rather than as a streaming program. We can write array programs by composing array operations together. The array programs, one for each individual query, can then be combined into a larger array program. Array programs can read from input arrays multiple times, and can produce intermediate and output arrays. The produced arrays can also be read from multiple times. We can execute array programs by dividing the operations into *passes*, and fusing all the operations in the same pass into a single loop. Each individual pass may contain operations from multiple queries, so we use the above process fusion method to fuse the array operations together. For array programs that require multiple passes, there are often many different ways to schedule the work among the passes. We perform *clustering* on the program to determine how many passes to perform, and how to schedule each array operation among the different passes. The choice of clustering affects runtime performance. To minimise the time spent reading and re-reading arrays, we would like to use a clustering which requires the minimum number of passes and intermediate arrays. Finding such a minimal clustering is NP-hard (Darte, 1999).

In Chapter 7, we find the minimal clustering by representing the array program's possible clusterings as an integer linear program, and using an external solver to find a solution that minimises our cost metric. Our clustering algorithm extends Megiddo and Sarkar (1997) to cluster array combinators, adding support for size-changing operations such as filter. The clustering algorithm of Megiddo and Sarkar (1997) uses integer linear programming to compute the clustering for an imperative loop nest rather than a set of array combinators. Individual

loops in the loop nest are assigned to clusters, and all loops in the same cluster are fused together. In Megiddo and Sarkar (1997), as with many imperative loop fusion systems, loops can only be fused together if their loop bounds are identical. In these systems, a loop that filters an array cannot be fused with a loop that consumes the filtered array, as they have different loop bounds. With process fusion, we *can* fuse a filter with its consumer, so a clustering algorithm for imperative loop nests would introduce more passes than necessary.

1.1 CONTRIBUTIONS

This thesis makes the following contributions:

MODAL TYPES TO ENSURE EFFICIENCY AND CORRECTNESS: if, due to time or cost constraints, we can only afford one pass over the input data, we need some guarantee that all our queries can be executed together. The streaming query language Icicle uses modal types to ensure all queries over the same input can be executed together in a single pass, as well as ensuring that the stream query has the same semantics as if it were operating over arrays. Icicle is described in Chapter 3.

PROCESS FUSION: a method for fusing stream combinators; the first that supports all three of multiple inputs, multiple concurrent queries, and user-defined combinators. In this streaming model, each combinator in each query is implemented as a sequential process. Together, the combinators of all queries form a concurrent process network. Processes are then fused together using an extension of synchronised product. Process fusion is described in Chapter 4.

FORMAL PROOF OF CORRECTNESS OF FUSION: a proof of correctness for process fusion, mechanised in the proof assistant Coq. The proof states that when two processes are fused together, the fused process computes the same result as the original processes. The proof is described in Section 4.6.

CLUSTERING FOR ARRAY-BACKED STREAMS: a clustering algorithm for array combinators, which supports size-changing operators such as filter. We allow size-changing operators to be assigned to the same cluster as operations that process their input and output arrays; existing clustering algorithms for imperative loop nests cannot assign size-changing operators to the same cluster as operations that consume the differently-sized

output array. Our algorithm encodes the clustering constraints of a set of array combinators as an integer linear program to be solved externally. The clustering algorithm is described in Chapter 7.

Before delving into these contributions, the next chapter introduces some background on different streaming models, as well as more concretely motivating why we want to execute multiple streaming queries concurrently.

CHAPTER 2

A BRIEF TAXONOMY OF STREAMING MODELS

We start by looking at how to write queries as *streaming programs*, so that we may query large datasets without running out of memory. Streaming programs consume data from their input streams element by element, processing the elements in sequential order, and need only store a limited number of elements at a time as local state. A streaming program cannot rewind an input stream to reread previous elements, or perform random access to read from an arbitrary element in the stream. These restrictions mean that a streaming program cannot, for example, sort all the input data in a single pass over the input stream, because single-pass sorting requires storing all the elements in memory. The upside of these restrictions is that if we can write our queries as streaming programs, we can be confident that they will run in constant space — no matter how large the input stream is. In general, input streams may be infinite, though in this thesis we focus on large but finite streams.

Streaming, as described above, is a rather general concept. This definition tells us what a streaming program is, but it does not offer any guidance on how to write streaming programs. In fact, there are many ways to write streaming programs; in this thesis we restrict our attention to streaming programs written in a *functional style*. The functional style of writing streaming programs involves using small stream transformers that are composed to create larger programs. The benefit of this style is that each stream transformer can be reasoned about and tested in isolation, without having to worry about any hidden dependencies or interference between different transformers.

There are numerous *streaming models* to choose from, and we must commit to a particular model before we can start writing programs. Choosing a streaming model requires making a trade-off between the performance overhead, which operations are supported, and the amount of extra bookkeeping the programmer must perform to write their program compared to a non-streaming implementation. We must compare different streaming models to make an informed decision. We start our initial comparison by focussing on two low-overhead streaming models to illustrate how they support different operations, and to motivate the use of Kahn process networks as a streaming model. In Chapter 5, we will compare with some more expressive but less efficient streaming models.

2.1 GOLD PANNING

Let us now describe a situation in which we would like to execute many queries over the same dataset. To avoid mixing up the details of streaming with the details of the example, we initially assume that the dataset fits in memory as a list, and write our queries as list programs. Throughout the thesis, we will refer back to this example as *gold panning*.

Suppose we have a file containing the historical prices for a particular corporate stock. The file contains many records; each record contains a date and the average price for that day, and all the records in the file are sorted chronologically. The records are stored on-disk in comma-separated values (CSV) format, and are represented in memory by the following Haskell datatype:

```
data Record = Record
 { time  :: Time
 , price :: Double }
```

We wish to appraise this stock to see whether it was, historically, a worthy investment. One quality of a good investment is that its price increases over time; we can quantify any increase by computing the linear regression of the price over time, using the coefficient of the line to approximate increase or decrease over time. It is very convenient to be able to summarise growth with one number, but stock prices rarely act as lines. While a line might be a good approximation for a stable stock price with few dips and bumps, it is a poor approximation for an unstable stock. Fortunately, we can use a statistical tool called the *Pearson correlation coefficient* to determine how linear the relationship is, and therefore how good the approximation is — which may be valuable information about the stock price in itself as well as denoting the confidence of our analyses. The Pearson correlation coefficient is defined as the covariance of price with time, divided by the product of the standard deviation of price and the standard deviation of time. We can also define the Pearson correlation coefficient geometrically: it is the cosine of the angle between the regression line of price over time, and the regression line of time over price.

Figure 2.1 shows the fluctuations of the example stock's price over a year, along with the regression line of price over time in red, and the regression line of time over price in blue. The stock price is far from a perfect line, but does show a clear upwards trend. In this graph, the correlation is represented by the angle between the red and blue regression lines; the smaller the angle between the two regression lines, the more closely correlated the two are, and the

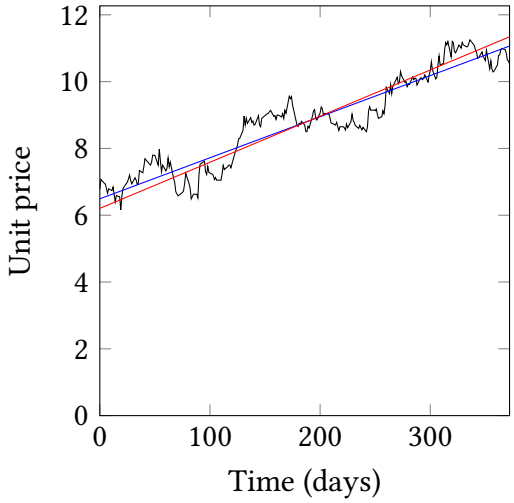


Figure 2.1: Analysis of a stock over a year

'straighter' the relationship is. The angle here corresponds to a correlation of 0.94, in a range from negative one to positive one.

Listing 2.1 contains the implementations of functions to compute the correlation coefficient, directly translated from the mathematical definitions. This implementation performs multiple passes over the input list, and is not numerically stable.

We can implement a more stable one-pass correlation algorithm using the covariance algorithm specified in Welford (1962). Although the details are quite complicated, we can express this algorithm as a fold over a list. The fold uses an initial state, `correlation_z`, and for each element updates the state with a worker function `correlation_k`. The one-pass correlation algorithm keeps track of the running means and standard deviations of both axes, which are used to compute the correlation. As such, the fold state contains more than just the correlation. After the fold has completed, we perform an *extraction* function, `correlation_x`, to extract the correlation from the state. Listing 2.2 contains the implementations of the fold worker functions for computing the correlation: `correlation_x`, `correlation_k` and `correlation_z`. With these worker functions, we can compute the correlation as follows:

```
correlation :: [(Double,Double)] → Double
correlation = correlation_x (foldl correlation_k correlation_z)
```

We can implement a function to compute the regression similarly; we omit the definitions of the regression worker functions. The definition of the function to compute the regression

```
correlation :: [(Double,Double)] → Double
correlation xys = covariance xys / stddev (map fst xys) * stddev (map snd xys)

stddev :: [Double] → Double
stddev xs = sqrt (mean (map (^2) xs) / mean xs * mean xs)

covariance :: [(Double,Double)] → Double
covariance xys =
 let xy = map (λ(x,y) → x * y) xys
 in mean xy - mean (map fst xys) * mean (map snd xys)

mean :: [Double] → Double
mean xs = sum xs / fromIntegral (length xs)
```

Listing 2.1: Multiple-pass correlation implementation

```
type State = (Double, Double, Double, Double, Double, Double)
correlation_z :: State
correlation_z = (0,0,0,0,0,0)

correlation_k :: State → (Double,Double) → State
correlation_k (mx, my, sd, sdX, sdY, n) (x,y) =
 let n'   = n   + 1
     dx   = x   - mx
     dy   = y   - my
     mx'  = mx  + (dx / n')
     my'  = my  + (dy / n')
     dx'  = x   - my'
     dy'  = y   - my'
     sd'  = sd  + dx * dy'
     sdX' = sdX + dx * dx'
     sdY' = sdY + dy * dy'
 in (mx',my',sd',sdX',sdY',n')

correlation_x :: State → Double
correlation_x (mx, my, sd, sdX, sdY, n) =
  let varianceX  = sdX / n
      varianceY  = sdY / n
      covariance = sd  / n
      stddevX = sqrt varianceX
      stddevY = sqrt varianceY
  in covariance / (stddevX * stddevY)
```

Listing 2.2: One-pass correlation implementation

uses the fold worker functions `regression_x` for extracting the final result, `regression_z` for the initial state, and `regression_k` to update the state for every input element:

```
regression :: [(Double,Double)] → Line
regression = regression_x (foldl regression_k regression_z)
```

Now that we have functions to compute the linear regression and the correlation, we can compute both at the same time. The following program returns a pair containing the correlation and regression:

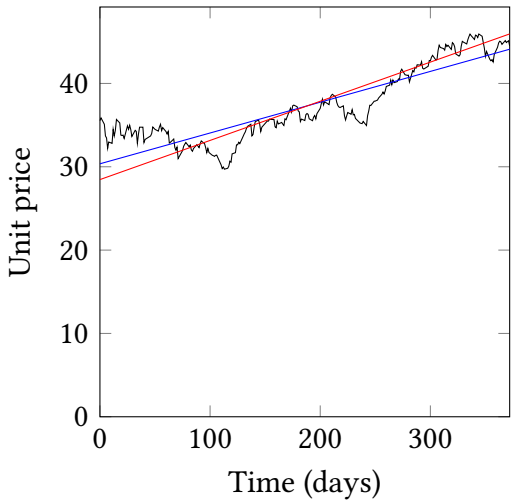
```
priceOverTime :: [Record] → (Line, Double)
priceOverTime stock =
  let timeprices = map (λr → (daysSinceEpoch (time r), price r)) stock
  in (regression timeprices, correlation timeprices)
```

Both `regression` and `correlation` functions take a list of pairs of numbers, so we first convert the `Record` values to pairs of numbers using the `map` combinator. Although this is a single program, it computes two values. Whether we think of this program as one query or two is inconsequential. The important part is that this program, as it is written, requires two traversals over the `timeprices` list. List programs can traverse the same list many times; in Section 2.2 we shall see how multiple traversals is a problem for streaming programs.

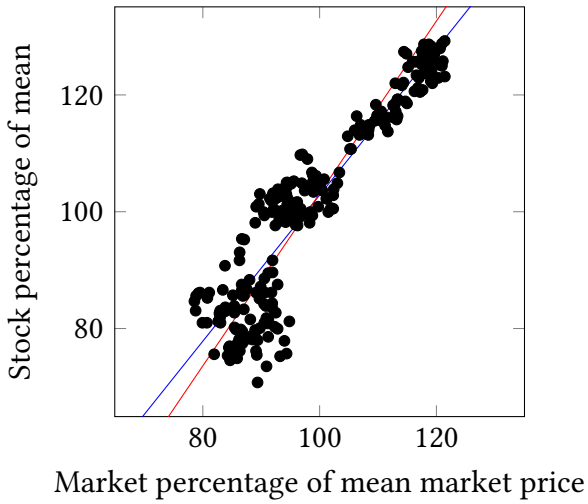
Stock prices rarely follow linear functions of time; even the best stocks go down once in a while, and sometimes the market as a whole can go down. Furthermore, even though this stock appears to be doing quite well if we consider it in isolation, we do not know whether it is an exceptional stock or an exceptional market. We are interested in comparing against the rest of the market as well.

To compare against the rest of the market, we have another file of records containing the average price of a representative subset of stocks. This representative subset is called a *market index*. We want to compare each day's price for our stock against the average price for the corresponding day in the index.

Figure 2.2a shows the linear regression and correlation of the market index price over time, while Figure 2.2b shows the linear regression and correlation of the stock price from Figure 2.1 compared to the market index price. The comparison of stock price to market index price compares each day's stock price against the corresponding day's market index price. Each day is visualised as the percentage difference between the day's stock price and the mean stock price for the period for the y axis, compared to the percentage difference between the day's market index price and the mean market index price for the period for the x axis. We can



(a) Market over time



(b) Stock over market

Figure 2.2: Analysis of a stock compared to market index

inspect the regression lines to see how the stock price tends to react to movement in the market index price. Both regression lines are slightly steeper than forty-five degrees, indicating that the stock price has grown faster than the market index price. The correlation, represented by the angle between the two regression lines, indicates a relatively strong linear relationship between the stock price and the market index price.

We can compute the comparison of stock price over market index price with the following program:

```
priceOverMarket :: [Record] → [Record] → (Line, Double)
priceOverMarket stock index =
  let joined = join (λs i   → time s `compare` time i) stock index
      prices = map  (λ(s,i) → (price s, price i))      joined
  in (regression prices, correlation prices)
```

The `join` operator matches each stock day against the corresponding market index day, and discards the data for any days missing from either input. We then extract both prices from the joined result and compute the regression and correlation. As with `priceOverTime`, this function requires two traversals of the `prices` list.

Since the analyses `priceOverTime` and `priceOverMarket` both provide useful information, we will perform both. It is just as easy to combine these queries together as it was to compute both the correlation and the regression. The following program computes both:

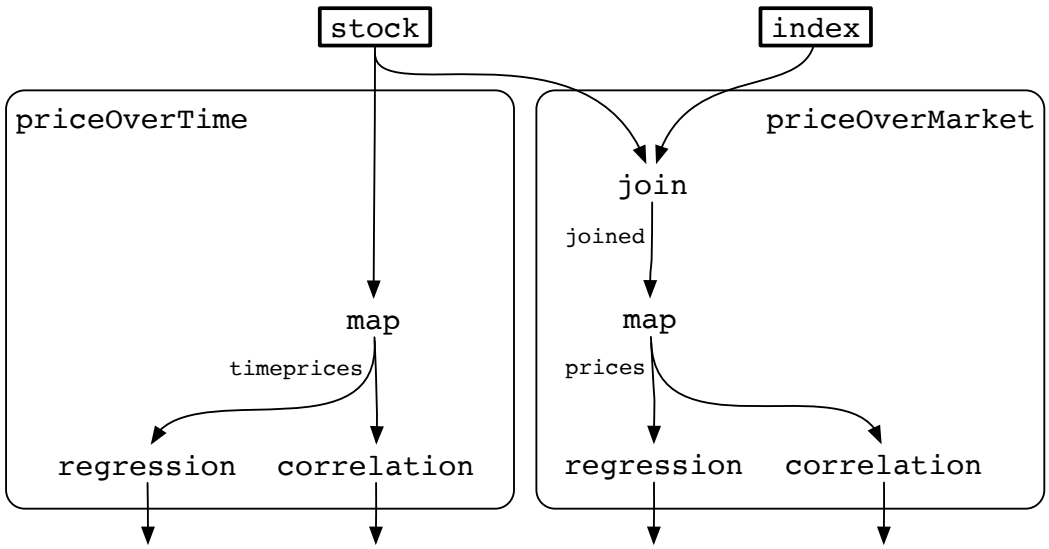


Figure 2.3: Dependency graph for queries `priceOverTime` and `priceOverMarket`

```
priceAnalyses :: [Record] → [Record] → ((Line, Double), (Line, Double))
priceAnalyses stock index =
  let pot = priceOverTime   stock
      pom = priceOverMarket stock index
  in (pot, pom)
```

Figure 2.3 shows the dependency graph for both queries. The nodes in this graph are the two input lists `stock` and `index`, and the list operators in each query. The two input lists are drawn inside boxes to distinguish them from the operator nodes. The edges are dependencies from one value to another; the `join` operator uses both the `stock` and `index` lists, so there are arrows from both `stock` and `index` to the `join` node. Edges that denote intermediate lists between two operators are labelled with the variable name used in the list program. Below the bottom of the graph, not shown, the results of the four `regression` and `correlation` operators are paired together to construct the return value. The large boxes bisecting most of the nodes denote which nodes are defined inside the `priceOverTime` function and which are defined in `priceOverMarket`. This dependency graph is a directed acyclic graph: the nodes `stock`, `timeprices`, and `prices` have multiple children; `joined` has multiple parents. Having multiple children means a list is mentioned multiple times, which generally corresponds to requiring multiple traversals of the list in a sequential evaluation.

Although a sequential evaluation of this list program requires multiple traversals of the input, we *can* rewrite it to be a single-pass streaming program. Our choice of streaming model dictates how difficult this rewrite will be.

2.2 PULL STREAMS

The first streaming model we look at are *pull streams*. Pull streams are used in the Volcano model of query execution (Graefe, 1989), where they are known as iterators; elsewhere, they can be known as cursors. The essence of a pull stream is that a consumer can *pull* on it to retrieve the next value from the producer. We represent a pull stream as a function with no parameters which either returns `Just` a value, or returns `Nothing` when the stream is finished. Since the function may need to read from a file or update some local state, it is expressed as an `IO` computation:

```
data Pull a = Pull (IO (Maybe a))
```

With this stream representation, we can implement analogues of the list combinators used in the example queries. We can map a function over a pull stream with the following definition:

```
map :: (a → b) → Pull a → Pull b
map a_to_b (Pull pull_a) = Pull pull_b
 where
  pull_b = do
    maybe_a ← pull_a
    return (case maybe_a of
              Nothing → Nothing
              Just a  → Just (a_to_b a))
```

Between unwrapping and wrapping the `Pull` constructor, the `map` function takes a function `pull_a` to compute the input stream values, and returns a function `pull_b` to compute the transformed stream values. Whenever the consumer of `map` calls `pull_b` to ask for the next value, `pull_b` in turn calls `pull_a` asking for the next value. When the input stream is finished, `pull_b` returns `Nothing` to denote that the output stream is also finished. Otherwise, `pull_b` applies the transform function `a_to_b` to the pulled element and returns the transformed element. In pull streams, consumers ask producers for the next value, and control flow bubbles up from consumer to producer.

We can also implement the fold combinator, `foldl`. Because pull streams can perform effects such as reading from a file, `foldl` for pull streams needs to be able to perform `IO` computations, as reflected in the result type:

```
foldl :: (b → a → b) → b → Pull a → IO b
foldl k z (Pull pull_a) = loop z
 where
  loop state = do
    maybe_a ← pull_a
    case maybe_a of
      Nothing → return state
      Just a → loop (k state a)
```

This implementation of `foldl` calls the local function `loop` with the initial state of `z`. The `loop` function repeatedly pulls from the pull function, `pull_a`, updating the state for every element in the input stream.

Consuming a stream is an effectful operation. Every time we call the pull function we get the next element, which means the pull function must somehow keep track of which value it is up to. For example, a pull function which reads from a file holds a file-handle, which in turn refers to some mutable state about the file offset. Every time we read from the file, the file offset is incremented. If two consumers were to ask the same pull function for the next element one after another, they would get different elements of the stream.

Listing 2.3 shows the type signatures of the pull stream versions of the folds we defined on lists earlier, `regression` and `correlation`, as well as the implementation of the `join` combinator. The `correlation` and `regression` functions can be implemented much like their list versions, using the pull implementation of `foldl`.

The `join` function executes by reading a value from each input stream and comparing the values using the given comparison function. Both input streams are sorted by some key, which the comparison function extracts and compares. If the keys are equal, `join` returns the pair. Otherwise, `join` pulls again from the input stream with the smaller key: since both streams are sorted by the key, if one stream has a higher key than the other, it means the stream with the higher key does not have a corresponding value for the smaller key. In our `priceOverMarket` example, the files are sorted by date and the comparison function compares the dates. We can join the two files in a streaming manner because both input files are already sorted by date; if the files were not sorted by date, we would need to perform a non-streaming join, for example a hash-join, which stores the entirety of one input in a hashtable in memory.

```
correlation :: Pull (Double,Double) → IO Double
regression  :: Pull (Double,Double) → IO Line

join        :: (a → b → Ordering) → Pull a → Pull b → Pull (a,b)
join comparekey (Pull pull_a) (Pull pull_b) = Pull (do
   a ← pull_a
   b ← pull_b
   go a b)
 where
  go (Just a) (Just b)
   = case comparekey a b of
      EQ → return (Just (a,b))
      LT → do
        a' ← pull_a
        go a' b
      GT → do
        b' ← pull_b
        go a b'
  go _ _ = return Nothing
```

Listing 2.3: Pull stream combinators

We cannot naively translate the list version of `priceOverTime` to use these streaming combinators, because the list version required multiple traversals. The following program will not compute the correct result because it uses the `timeprices` stream twice:

```
priceOverTime_pull_bad :: Pull Record → IO (Line, Double)
priceOverTime_pull_bad stock = do
  let timeprices = Pull.map (λr → (daysSinceEpoch (time r), price r)) stock
  r ← Pull.regression  timeprices
  c ← Pull.correlation timeprices
  return (r, c)
```

Computing the regression pulls all the values from the `timeprices` stream and folds over them until the stream is exhausted. After computing the regression, the program computes the correlation of the same input stream. When the correlation tries to read the `timeprices` stream again, the stream has already been exhausted. For this reason, we say that pull streams cannot be used multiple times. This single-use restriction also exists in other streaming models, such as Java 8 Streams (Reese, 2014) and Strymonas (Biboudis, 2017). Some streaming models, such as Iteratees (Kiselyov, 2012), do not explicitly state any single-use restriction, but exhibit the same potentially surprising behaviour when multiple consumers compete for elements from

the same stream. In Section 2.4 we shall see a streaming model that is more strict and requires that streams be used exactly once and cannot be discarded, a property commonly known as *linearity*.

Since our input streams are sourced from files stored on disk, we could imagine *rewinding* the input streams and re-reading the files again from the start. If we were to rewind the stream, all the effects and all the work that went into computing the stream the first time would have to be done a second time: for this example, we would at least need to parse each line in the file again. Rewinding would allow this program to compute the correct result, but it is only feasible for *persistent* input streams. Real-time inputs such as sensor data may accumulate too quickly to be stored indefinitely, and for network-backed storage, the extra communication time may make performing multiple reads impractical.

Fortunately, because `regression` and `correlation` are both computed by folds, we can combine the two into a single fold. In the following program, the fold worker function `both_k` and initial state `both_z` compute both regression and correlation at the same time:

```
regressionCorrelation_pull :: Pull (Double,Double) → IO (Line, Double)
regressionCorrelation_pull stream = do
  (r,c) ← Pull.foldl both_k both_z stream
  return (regression_x r, correlation_x c)
 where
  both_k (r,c) v = (regression_k r v, correlation_k c v)
  both_z         = (regression_z,     correlation_z)

priceOverTime_pull :: Pull Record → IO (Line, Double)
priceOverTime_pull stock = do
  let timeprices = Pull.map (λr → (daysSinceEpoch (time r), price r)) stock
  regressionCorrelation_pull timeprices
```

This program computes the correct value in a single traversal of the input stream. To write this version, we have had to manually look inside the definitions of `correlation` and `regression` and duplicate them. This was relatively easy because both use-sites were folds. This process of combining two folds into one is a simple instance of a transform known as *tupling*. Transforms such as (Hu et al., 1997, 2005; Chiba et al., 2010) can automatically perform tupling for some programs, but do not support combinators with multiple input streams such as `join` or `append`. We discuss tupling further in Section 6.2.

Let us turn our attention to the second query, `priceOverMarket`. We can use the same function `regressionCorrelation_pull` that we used above, like so:

```
priceOverMarket_pull :: Pull Record → Pull Record → (Line, Double)
priceOverMarket_pull stock index =
  let joined = Pull.join (λs i   → time s `compare` time i) stock index
  let prices = Pull.map  (λ(s,i) → (price s, price i))      joined
  regressionCorrelation_pull prices
```

We now have pull stream implementations of both `priceOverTime` and `priceOverMarket`, but when we wish to compute both at the same time, we cannot simply pair them together as we did in the list implementation of `priceAnalyses` — this time because the `stock` stream is mentioned multiple times.

When we implemented the pull stream version of `priceOverTime`, we had to look at the two occurences where the `timeprices` stream had been used. We had to inline both places where the stream was used and manually write a new function to do the work of both. Both were fairly simple folds. Doing the same for `priceAnalyses` is more complicated: we would need to implement a special version of the `join` combinator used inside `priceOverMarket`, which not only joins the two input streams together, but also computes the regression and correlation of its stock stream at the same time.

It might appear that, since the `joined` stream contains pairs from both `stock` and `index`, we could use this to compute the correlation and regression of the the `stock` component alone. Such a query would be easier to combine with `priceOverMarket`, but this query would compute a different result, since the `joined` stream only contains elements from `stock` for which corresponding days exist in the `index` stream.

Pull streams are not helping us execute multiple queries at a time. If we wish to execute multiple queries in a single-pass, we need to be able to mention streams multiple times. To execute these shared streams, each time we read from a shared stream, we need some way to distribute this element among all of the shared stream's consumers.

2.2.1 *Streaming overhead*

The pull stream representation can incur some overhead due to the need to wrap elements in `Maybe` constructors. For simple combinators such as map, however, the overhead can be optimised away by performing program transformation. Consider the following function, which applies two worker functions to the elements in a stream:

```
map2 :: (a → b) → (b → c) → Pull a → Pull c
map2 f g stream_a
 = let stream_b = Pull.map f stream_a
       stream_c = Pull.map g stream_b
   in  stream_c
```

We could write this program in an equivalent way by composing the two functions together and performing a single map: `Pull.map (g ∘ f) stream_a`. Fortunately, after some optimisation, both programs incur the same amount of overhead. To demonstrate concretely the overhead of composing stream transformers, we take the definition of `Pull.map` and inline it into the use-sites in `stream_b` and `stream_c` above. After removing some wrapping and unwrapping of `Pull` constructors, we have the following function:

```
map2 f g (Pull pull_a) = Pull pull_c
 where
  pull_b = do
    a ← pull_a
    return (case a of
              Nothing → Nothing
              Just a' → (Just (f a')))
  pull_c = do
    b ← pull_b
    return (case b of
              Nothing → Nothing
              Just b' → (Just (g a')))
```

When we pull from `pull_c`, it asks `pull_b` for the next element, which in turn asks `pull_a`. When there is a stream element to process, `pull_a` constructs a `Just` containing the stream element and returns it to `pull_b`. This `Just` constructor is then destructed by `pull_b` so the function `f` can be applied to the element, before wrapping the result in a new `Just` which is returned to `pull_c`. Now, `pull_c` must perform the same unwrapping and wrapping on the returned value, even though we statically know that whenever `pull_a` returns a `Just`, `pull_b` must also return a `Just`.

To take advantage of our static knowledge and remove the superfluous wrapping and unwrapping, we first transform the program by inlining `pull_b` into where it is called in `pull_c`. Then, using the monad laws, we can rewrite the `return` statement containing the case expression from `pull_b`, nesting this case expression inside the scrutinee of the other case expression:

```
map2 f g (Pull pull_a) = Pull pull_c
 where
  pull_c = do
    a ← pull_a
    return (case (case a of
                   Nothing → Nothing
                   Just a' → (Just (f a')))
              Nothing → Nothing
              Just b' → (Just (g b')))
```

The nested case expression returns statically-known constructors of `Nothing` or `Just`, which the outer case expression immediately matches on. We remove the intermediate step using the *case-of-case* transform (Jones and Santos, 1998), which converts these nested case expressions to a single case expression:

```
map2 f g (Pull pull_a) = Pull pull_c
 where
  pull_c = do
    a ← pull_a
    return (case a of
              Nothing → Nothing
              Just a' → (Just (g (f b))))
```

By applying some standard program transformations, we have combined the two maps into one, and we have removed the overhead of additional constructors. Optimising compilers perform similar transforms as part of their suite of general purpose optimisations (Jones, 1996).

For other operations, the streaming overhead can be harder to remove. Listing 2.4 shows the implementation of the `filter` combinator. The function `pull_a'` pulls an element and checks whether it satisfies the given predicate. If the element does not satisfy the predicate, `pull_a'` performs a recursive loop to check the next element. The recursion in this function makes it harder for the compiler to inline and optimise, and the overhead may not be removed. Stream fusion (Coutts et al., 2007), which uses a more sophisticated pull stream representation, allows pull functions to return a *skip* constructor in place of filtered-out elements, instructing the consumer to pull again. The stream fusion implementation of `filter` instructs the consumer to pull the next element when the predicate fails instead of recursively looping, and is easier to inline and optimise. Compilers that use an intermediate language based on sequent calculus may be able to inline and optimise recursive functions directly, without needing to

```
filter :: (a → Bool) → Pull a → Pull a
filter predicate (Pull pull_a) = Pull pull_a'
 where
  pull_a' = do
    a ← pull_a
    case a of
     Nothing → return Nothing
     Just a'
      | predicate a' → return (Just a')
      | otherwise    → pull_a'
```

Listing 2.4: Pull implementation of `filter`

convert stream operations to non-recursive functions (Maurer et al., 2017). We discuss other representations of pull streams in Section 6.1. These other representations of pull streams, as well as the representation used by stream fusion, do not afford extra expressivity over the pull streams described above; the same set of combinators can be implemented with the same asymptotic space and time behaviour, but with improved constant factors.

2.3 PUSH STREAMS

Push streams are the conceptual dual of pull streams: rather than the consumer pulling elements from the producer, in push streams the producer pushes elements to the consumer. As we shall see, the advantage of push streams is that they enable stream elements to be shared among multiple consumers: a producer can push the same value to multiple consumers. This sharing of elements makes it easier to perform multiple queries over the same input stream.

A push stream is a function which accepts a (`Maybe a`) and performs some `IO` effect, for example writing to a file, or writing to some mutable state. This could be represented by the type (`Maybe a → IO ()`), which is the dual of the pull stream (`IO (Maybe a)`). However, this representation provides no direct way to retrieve a result from a consumer: for example, the return value of our correlation or regression. This is a common enough use-case that it justifies a departure from the conceptual clarity of using the exact dual. We instead use the following representation:

```
data Push a r = Push
  { push :: a → IO ()
  , done :: IO r }
```

We augment the definition with an extra type parameter, `r`, for the result type. Since the result only becomes available at the end of the stream, we separate the two cases of the (`Maybe a`) argument into two functions, `push` and `done`. When the producer has an element to give to the consumer, the producer calls `push`. When the stream is finished, the producer calls `done` to retrieve the result.

In this representation, it is the consumers that are values of type (`Push a r`): they are sinks into which we can push values of type `a`, and eventually get an `r` back. This inversion of control for pull streams leads to a fundamental difference in how we program with push streams, and what we can express with push streams.

The push streams described here are analogous to the *sinks* described in Bernardy and Svenningsson (2015) and Lippmeier et al. (2016), albeit with a slightly different representation. In this thesis, we use the push and pull terminology of Kay (2009). However, the 'push' in 'push streams' is different from the 'push' in the 'push model' used for database execution, as described in Neumann (2011). In the *Neumann push model*, a stream producer is represented as a continuation which takes a sink to push values into. Once the consumer provides a sink, the producer repeatedly pushes all its values to the provided sink. The control-flow for the Neumann push model is the same as for *push arrays*, as described in Claessen et al. (2012). Like pull streams, the Neumann push model does not support executing multiple queries concurrently; unlike pull streams, the Neumann push model does not support combinators with multiple inputs except append. We discuss the Neumann push model in more detail later in Section 6.3.

We cannot map a function over the elements in push streams in the way that we would with lists or pull streams, because the definition of (`Push a r`) uses the stream element type `a` as a function argument, rather than as a function result. Instead, we implement contravariant-map, or `contramap`, like so:

```
contramap :: (a → b) → Push b r → Push a r
contramap a_to_b bs = Push push_a done_a
 where
  push_a a = push bs (a_to_b a)
  done_a   = done bs
```

The `contramap` function takes a function to convert values of type `a` to values of type `b` and a sink to push values of type `b` to, returning a sink which can receive values of type `a`. When a producer tries to push an input element into the returned stream, the `push_a` function converts this element to a value of type `b` and pushes it further on to the consumer of `b`. Unlike with

consumers of pull streams, a push consumer has no way of choosing among multiple inputs. In the push stream model, the producer is in control while the consumer passively waits for its next input value.

We *do* in fact have a regular (covariant) map function for push streams, but this transforms the result returned at the end of the stream, rather than the input stream elements. We map a function over the result like so:

```
map_result :: (r → r') → Push a r → Push a r'
map_result r_to_r' push_a = Push (push push_a) done_a'
 where
  done_a' = do
    r ← done push_a
    return (r_to_r' r)
```

The type of `foldl` for push streams is similar to pull streams, except instead of taking the pull stream to read from, it returns a push stream which will eventually return the result. The return value is an `IO` computation containing the push stream because we use a mutable reference to store the current state, which must be allocated before returning the stream. As values are pushed into the sink, the mutable reference containing the fold state is updated with the current result of the fold:

```
foldl :: (b → a → b) → b → IO (Push a b)
foldl k z = do
  ref ← newIORef z
  let push_a a = do
       state ← readIORef ref
       writeIORef ref (k state a)
  let done_a = readIORef ref
  return (Push push_a done_a)
```

As with the list and pull stream implementations, we can use this `foldl` function to implement `correlation` and `regression`.

In order to share a stream between multiple consumers, we need some way to broadcast messages and push each element to many consumers. We can broadcast to two consumers by combining two consumers into one before connecting it to a producer. The following function, `dup_ooo`, duplicates a stream among two consumers, and returns a pair containing both results. We call this operation `dup_ooo` because it *dup*licates elements into two *o*utput sinks (push streams), returning a new *o*utput sink; the reason for this name will become apparent when

we see other ways to duplicate streams in Section 2.4. The implementation of `dup_ooo` is as follows:

```
dup_ooo :: Push a r → Push a r' → Push a (r,r')
dup_ooo a1 a2 = Push push_a done_a
 where
  push_a a = do
    push a1 a
    push a2 a

  done_a = do
    r  ← done a1
    r' ← done a2
    return (r, r')
```

We could also use the applicative functor (McBride and Paterson, 2008) instance for push streams to combine consumers together, specifying how to transform and combine the results. The applicative functor implementation is similar to the `dup_ooo` function specified above. This `dup_ooo` function could then be written equivalently as (`dup_ooo a1 a2 = (,) <$> a1 <*> a2`).

We can use `dup_ooo` and `contramap` to implement `unzip`, which deconstructs a stream of pairs into a pair of streams:

```
unzip :: Push a r → Push b r' → Push (a,b) (r,r')
unzip push_a push_b = dup_ooo (contramap fst push_a) (contramap snd push_b)
```

Pairs of `a` and `b` flow from the returned push stream into the argument streams. When the stream is over, the results are paired together and flow from the argument streams to the returned stream. This inverted control flow is because the stream representation makes operations on the elements contravariant, while operations on the stream results are covariant.

With these combinators, we can write the `priceOverTime` query using push streams:

```
priceOverTime_push :: IO (Push Record (Line,Double))
priceOverTime_push = do
  reg   ← Push.regression
  cor   ← Push.correlation
  let cm = Push.contramap
    (λr → (daysSinceEpoch (time r), price r))
    (Push.dup_ooo reg cor)
  return cm
```

This program computes both correlation and regression in a streaming fashion. In comparison to the list version of `priceOverTime`, we have explicitly combined both consumers and reversed the control flow.

Unfortunately, we cannot implement a streaming version of `priceOverMarket` with push streams alone, because it requires joining two input streams by date. Recall the `join` combinator, which takes two input streams and retrieves a value from each. At every step the `join` combinator chooses which stream to pull from, pulling from the stream with the smaller value. With push streams, a consumer cannot choose which input stream to pull from, or when: the consumer is a function waiting to be called with its input elements, always ready to accept elements as they come.

This inability to join two streams by date is one instance of a more general limitation of push streams. Push streams also cannot implement `zip`, which pairs two inputs together, because the consumer needs to control the computation to alternate between each input. Except for one special case, push streams do not support combinators with multiple inputs. The special case is that a push stream can react to multiple inputs in the order they are received. As a list program, this is similar to taking two lists and at each step non-deterministically choosing which list to pull an element from. In certain circumstances we can control the push order and use this merge to append two streams. Because the push order is controlled outside of the merge, appending two streams in this way separates the append logic from the merge combinator which defines the appended stream.

We shall see more examples of push programs in Chapter 3.

2.4 POLARISED STREAMS

Stream sharing allows push streams to support multiple queries by broadcasting the elements to multiple consumers, but they do not support streaming operators with multiple inputs, except for non-deterministic merge; pull streams support operators with multiple inputs, but they do not support multiple queries (Kay, 2009). Combining pull and push streams together in the same program, in the form of *polarised streams*, allows us to support multiple inputs and multiple queries (Lippmeier et al., 2016).

Although we cannot share the elements of a pull stream among multiple pull consumers, we can share the elements of a pull stream among one push consumer and one pull consumer. We call this operation `dup_ioi` because it *dup*licates an *i*nput source (pull) into an *o*utput sink (push), returning a new *i*nput source (pull):

```
dup_ioi_ignore_result :: Pull a → Push a r → Pull a
dup_ioi_ignore_result (Pull pull_a) push_b = Pull pull_a'
 where
  pull_a' = do
    v ← pull_a
    case v of
     Nothing → do
      _ ← done push_b
      return Nothing
     Just a → do
      push push_b a
      return (Just a)
```

We achieve this duplication by constructing a new pull stream which, when pulled on, pulls an input element from its source `pull_a`. The pulled element is pushed into the sink `push_b`, before being returned to the caller as an element of the constructed pull stream. In this implementation, the result of the push stream is ignored because the pull stream representation has no way to return a result at the end of the stream.

Encoding the result of a stream inside the stream itself is not important for single-consumer pull streams, because it is usually the consumer of the stream that computes the result. When mixing stream representations to allow multiple consumers, however, we need to be able to capture the result of all the consumers. We extend the pull stream representation so that instead of returning a `Maybe` and using `Nothing` to signal the end of the stream, streams now return an `Either` with (`Left a`) to signal an element and (`Right r`) to signal the result at the end of the stream:

```
data PullResult a r = PullResult (IO (Either a r))
```

With this extended pull stream representation, we can implement a version of `dup_ioi` that keeps the results of the input stream and the output stream, and pairs them together. This version of `dup_ioi` is shown in Listing 2.5.

Modifying `dup_ioi_ignore_result` to work on the new representation only required changing the constructors for the stream and adding the return value; other combinators are modified similarly. We use the same naming convention for suffixes, for example `map_i` for mapping pull streams, and `map_o` for contravariantly mapping push streams. When consuming a pull stream by folding over it, we return the fold result as well as the stream result. The type sig-

```
dup_ioi :: PullResult a r → Push a r' → PullResult a (r,r')
dup_ioi (PullResult pull_a) push_b = PullResult pull_a'
 where
  pull_a' = do
    v ← pull_a
    case v of
     Right r → do
      r' ← done push_b
      return (Right (r,r'))
     Left a → do
      push push_b a
      return (Left a)
```

Listing 2.5: Polarised implementation of `dup_ioi`

nature for `foldl_i` changes to include the stream result; the implementation change is similar to the change for `dup_ioi`:

```
foldl_i :: (b → a → b) → b → PullResult a r → IO (b,r)
```

We can also copy elements from a pull stream to a push stream: we call this operation *draining* the pull stream. To drain a stream, we loop over the elements in the pull stream and push each one into the push stream. At the end, we return a pair of the results of both streams:

```
drain_io :: Pull a r → Push a r' → IO (r, r')
drain_io (Pull pull_a) push_a = loop
 where
  loop = do
    v ← pull_a
    case v of
     Left a → do
      push push_a a
     Right r → do
      r' ← done push_a
      return (r, r')
```

We can combine `drain_io` and `dup_ooo` together to duplicate a pull stream into two push streams, which we call `dup_ioo`:

```
dup_ioo :: Pull a r → Push a r' → Push a r'' → IO (r,(r',r''))
dup_ioo pull0 push1 push2 = drain_io pull0 (dup_ooo push1 push2)
```

```
join_iii :: (a → b → Ordering) → PullResult a r
         → PullResult b r'      → PullResult (a,b) (r,r')
join_iii comparekey (PullResult pull_a) (PullResult pull_b) = PullResult (do
   a ← pull_a
   b ← pull_b
   go a b)
 where
  go (Left a) (Left b)
   = case comparekey a b of
      EQ → return (Left (a,b))
      LT → do
        a' ← pull_a
        go a' b
      GT → do
        b' ← pull_b
        go a b'
  go (Right a) (Right b) = return (Right (a,b))
  go (Left _) (Right b) = do
    a' ← pull_a
    go a' (Right b)
  go (Right a) (Left _) = do
    b' ← pull_b
    go (Right a) b'
```

Listing 2.6: Polarised implementation of `join_iii`

We can also implement `dup_oii` by flipping the arguments of `dup_ioi`, and compose the various `dup` functions together to duplicate an arbitrary number of outputs. With `dup_ioi`, `dup_oii`, `dup_ioo` and `dup_ooo`, we can duplicate a stream when there is no more than one pull consumer. Joining multiple input streams together, as in the `join` combinator, is the dual: there can be no more than one push producer. Recall that the `join` combinator required both inputs to be pull streams, and could not be implemented with push streams alone. With the polarised naming convection, this version of `join` is called `join_iii`. Because the input streams have result values, we ensure that the joined stream's result contains the results of both inputs. To compute both results, when one stream ends before the other we drain the unfinished stream until we reach the result. Other than this draining, the implementation of `join_iii` shown in Listing 2.6 follows the implementation of `join`.

We can also join two streams when one is a pull stream and the other is a push stream: this is called `join_ioo`. Conceptually, this combinator has an input pull stream of type `a` and an input push stream of type `b`, with an output push stream of pairs of `a` and `b`. The definition

```
join_ioo :: (a → b → Ordering) → PullResult a r
         → Push (a,b) r'        → Push b (r,r')
join_ioo comparekey (PullResult pull_a) push_ab = Push push_b done_b
 where
  push_b b = do
    a ← pull_a
    case a of
     Left a' → case comparekey a b of
      EQ     → push push_ab (a,b)
      LT     → push_b b
      GT     → return ()
     Right _ → return ()
  done_b = do
    a ← pull_a
    case a of
     Left _   → done_b
     Right a' → do
      b ← done push_ab
      return (a,b)
```

Listing 2.7: Polarised implementation of `join_ioo`

for `join_ioo` is given in Listing 2.7. The output push stream is given as an argument while the input push stream is the return value.

In the implementation of `join_ioo`, the returned push stream accepts values of type `b`. When a new value is pushed, it repeatedly reads values from the input pull stream until the pulled value is equal to or greater than the pushed value using the given ordering function to compare the keys. When the ordering function says the two keys are equal, it pushes the pair to the output stream. When the pull stream ends before the push stream, this implementation reads the end of the pull stream multiple times; the pull stream always returns the stream result after the end of the stream. Other multiple-input combinators can be inverted similarly to support pull-push-push (`_ioo`) and push-pull-push (`_oio`) versions.

We can implement both `priceAnalyses` queries — `priceOverTime` and `priceOverMarket` — by mixing pull and push streams. We assign a polarity of push or pull to all streams in both queries, starting with the input streams. Figure 2.4 shows the dependency graph with polarised combinators and explicit duplications. The polarity of each stream is depicted as a filled or unfilled circle. Filled circles • represent pull streams because they always contain the next element or the result. Unfilled circles ∘ represent push streams because they are a hole into which elements can be pushed.

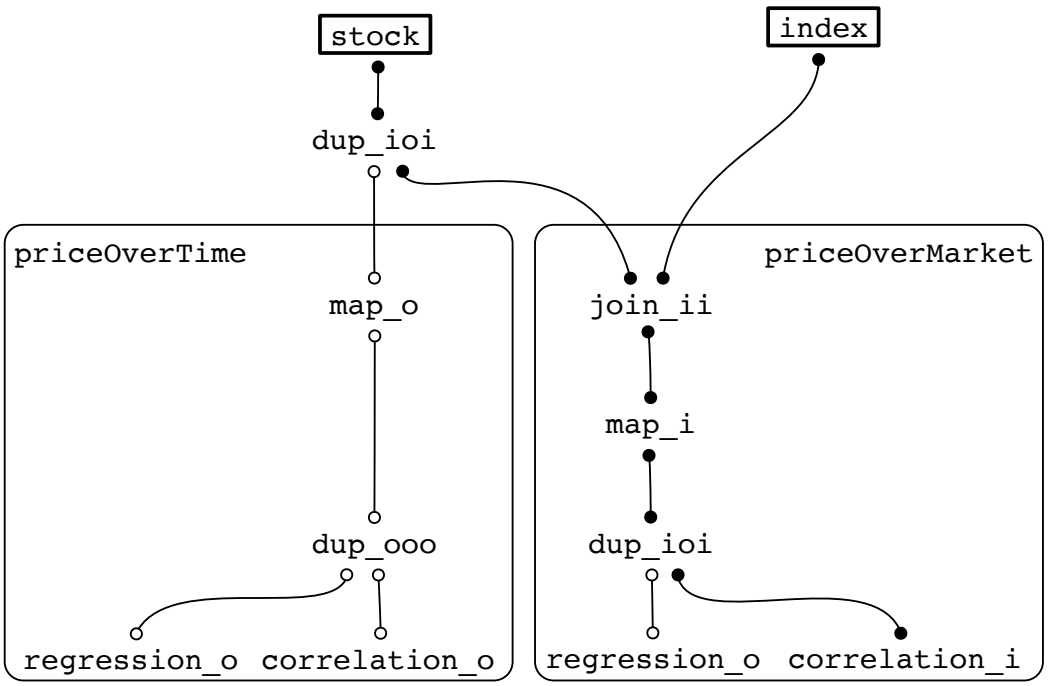


Figure 2.4: Polarised dependency graph for `priceOverTime` and `priceOverMarket`

We begin by classifying both input streams as pull streams: we can copy elements from pull stream producers into push stream consumers, but not vice versa. Classifying inputs as pull streams allows the most flexibility, as consumers of the inputs can still be classified as either pull or push operators. The `stock` input stream is used twice, so at least one of the use-sites must be classified as push. Since we were able to express `priceOverTime` entirely as a push stream computation, we duplicate `stock` into a push stream for `priceOverTime` and a pull stream for `priceOverMarket`. For `priceOverMarket`, we can join both pull streams and map over it. To compute both regression and correlation, one of the folds must be a push; in this case either consumer can be pull or push. The decision is inconsequential. There are many different polarity assignments for this program, but they all compute the same result.

Listing 2.8 shows the implementation of this polarised dependency graph for both queries. This streaming single-pass implementation requires more complex control-flow than the list version. Stream elements flow "backwards" from function result to argument in the places where push streams are used, and flow "forwards" where pull streams are used. When a pull stream is duplicated into a push stream, the stream results for the push stream are nested inside the resulting pull stream; recovering these results involves pattern-matching on the nested tuple.

```
priceOverTime_o :: IO (Push Record (Line,Double))
priceOverTime_o = do
  pot_regres    ← regression_o
  pot_correl    ← correlation_o
  let folds      = Push.dup_ooo reg cor
  let timeprices = map_o (λr → (daysSinceEpoch (time r), price r)) folds
  return timeprices

priceOverMarket_ii :: PullResult Record r → PullResult Record r'
                   → IO (Line,(Double,(r,r')))
priceOverMarket_ii stock index = do
  let joined  = join_iii (λs i   → time s `compare` time i) stock index
  let prices  = map_i    (λ(s,i) → (price s, price i))      joined
  pom_regres ← regression_o
  let prices' = dup_ioi prices pom_regres
  correlation_i prices'

priceAnalyses_ii :: PullResult Record r → PullResult Record r'
                 → IO ((Line,Double), (Line, Double))
priceAnalyses_ii stock index = do
  pot        ← priceOverTime_o
  let stock'  = dup_ioi stock pot
  result     ← priceOverMarket_ii stock' index
  case result of
    (potC,(potR,((r,(pomC,pomR)),r'))) →
      return ((potC,potR), (pomC,pomR))
```

Listing 2.8: Polarised implementation of `priceOverTime` and `priceOverMarket`

```
zip_ioo :: PullResult a r → Push (a,b) r' → Push b (r,r')
zip_ioo (Pull pull_a) push_ab = Push push_b done_b
 where
  push_b b = do
   a ← pull_a
   case a of
    Left a' → push push_ab (a',b)
    Right _ → return ()
  done_b = do
   a ← pull_a
   case a of
    Left _  → done_b
    Right r → do
     r' ← done push_ab
     return (r,r')
```

Listing 2.9: Polarised implementation of `zip_ioo`

Assigning polarities is a global analysis, in that we need to inspect the dependency graph containing all queries, rather than looking at each query or each combinator in isolation. If we add a new query to `priceAnalyses`, we need to consider the existing polarities when assigning polarities to the new query. Suppose we have an *industry index* which, like the market index, contains average prices of a representative subset of stocks. For lists, we can reuse the `priceOverMarket` query to compute how closely our stock follows the industry. For polarised streams, we cannot reuse `priceOverMarket_ii` in `priceAnalyses_ii` to compare the stock against both indices in a single traversal, because this would require duplicating the stock stream into two pull consumers. We need to implement another version of the same query with different polarities: `priceOverMarket_oi`. Polarised streams are not composible and can require code duplication.

2.4.1 *Diamonds and cycles*

Recall that the list definition of `correlation` takes a list of pairs of doubles and performs a fold over them. We could write another version that takes two lists of doubles and then *zips* the two lists together before performing the fold. When correlating values from the same input list, the list-of-pairs version requires one fewer intermediate list; when correlating values from different lists, the pair-of-lists version is slightly more convenient. Which version is preferable depends on the situation, but for lists, the difference is usually minor.

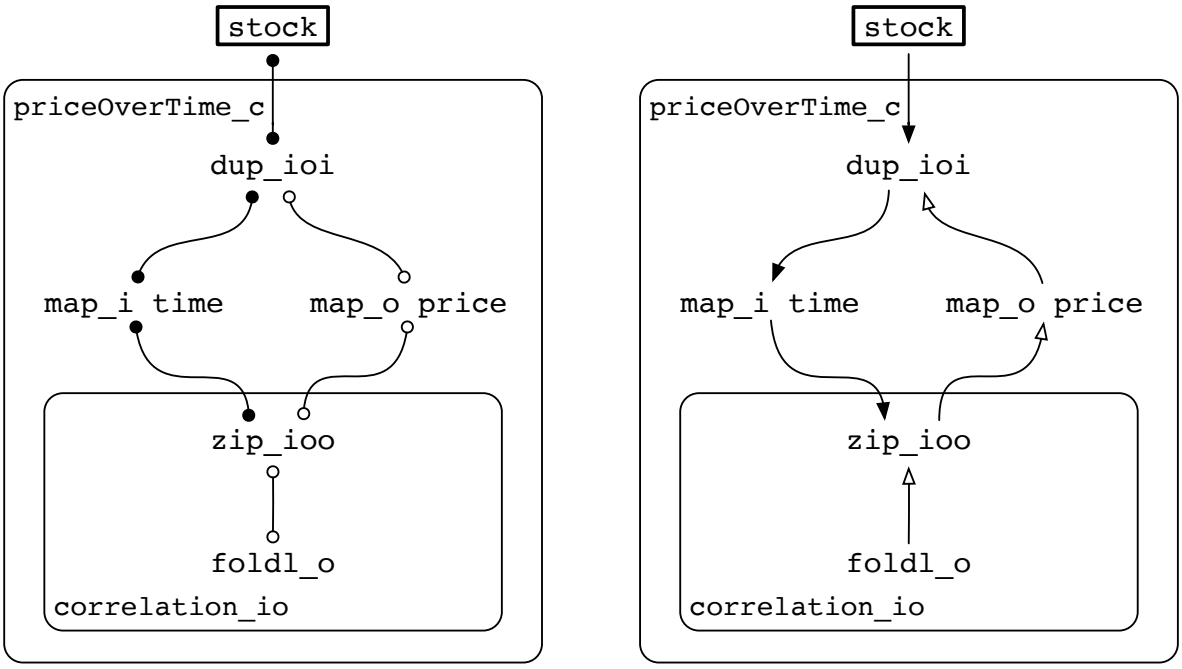


Figure 2.5: Polarised dependency graph with diamond (left), and control-flow graph (right)

With polarised streams, we cannot execute `priceOverTime` with the pair-of-lists version, although we can assign polarities. The lefthand side of Figure 2.5 shows the polarised graph for a hypothetical version of `priceOverTime` that only computes the correlation and uses zip. The `zip_ioo` combinator, implemented in Listing 2.9, is similar to the `join_ioo` combinator; it has an input pull, an input push, and an output push. When the input push stream receives a value, `zip_ioo` reads from the pull stream and sends the pair to the output push stream.

Although it is not obvious from the polarised dependency graph, the combinators have a recursive dependency on each other. The polarised diagram shows *elements* flowing down, but the *control flow* for push streams is upwards. The righthand side of Figure 2.5 shows the control flow for the polarised diagram, after replacing the edges corresponding to pull streams with filled downward arrows and replacing the edges corresponding to push streams with unfilled upward arrows. In this graph, the recursive dependency is illustrated by the cycle between `dup_ioi`, `zip_ioo` and the two maps. This cycle complicates translating the graph to an implementation.

The incomplete implementation in Listing 2.10 also demonstrates the recursive dependency between the four operators. The `priceOverTime_c` function defines a set of stream transformers, but we have no way to execute this stream transformer and extract the result. The incomplete implementation constructs a stream transformer, but does not directly compute the stream

```
priceOverTime_c :: PullResult Record r → IO (Double, r)
priceOverTime_c stock =
  let stock' :: PullResult Record (Double,r)
             = dup_ioi stock prices
      times  :: PullResult Record (Double,r)
             = map_i time stock'
      prices :: Push Record Double
             = map_o price cor
      cor    :: Push Record Double
             = correlation_io times
  in _ -- incomplete: no way to run computation

correlation_io :: PullResult Double r → Push Double (Double, r)
correlation_io stream_a = zip_ioo stream_a correlation_o
```

Listing 2.10: Incomplete polarised implementation of `priceOverTime_c`

result. All the combinators are *passive combinators*, constructing a stream transformer that responds to push or pull requests rather than actively pulling or pushing. *Active combinators* like `drain_io` and `foldl_i` actively consume pull streams rather than transforming them, and return an `IO` action containing the stream result. Without an active combinator, the query will not execute. Active combinators can consume pull streams and output to push streams. Active combinators cannot actively consume push streams, because the control flow for push streams is driven by the producer. Similarly, they cannot actively produce pull streams. None of the combinators in this query can be implemented as active combinators because they all consume push streams or produce pull streams. Instead we must hand-optimise the program, combining the duplicate, maps and zip into one combinator, as in the original version of `priceOverTime`.

2.5 KAHN PROCESS NETWORKS

The three streaming models we have seen — pull, push, and polarised streams — differ essentially in what drives the computation. With pull streams, the consumer drives the computation. With push streams, the producer drives the computation. With polarised streams, the active combinator — which may be a pull consumer, a push producer, or even both — drives the computation. All these systems have one operator driving the computation. When we have multiple queries to execute, it can be hard to choose just one combinator to drive the entire computation.

An alternate streaming model, which allows many operators to control the computation and supports executing multiple queries, is a *Kahn process network*. A Kahn process network is a concurrent process network with restrictions to ensure deterministic execution. Each combinator inside each query becomes a communicating process in the network. Processes communicate through input channels from which they can pull values, and output channels to which they can push values. Each process can have multiple inputs and outputs, and the process chooses the order to pull from its inputs and push to its outputs.

Concurrent programs can be hard to write and debug because the *schedule*, which specifies the interleaving of process execution, depends on environmental factors outside the program itself — for example, the number of physical processors available, and which other processes are also being executed. Because the environment is not controlled by the processes themselves, we say the schedule is chosen *non-deterministically*. Likewise, if a program gives different results for different schedules, we say the result is *non-deterministic*. Kahn process networks are a restricted form of static process network where program values cannot be affected by the schedule (Kahn et al., 1976): the schedule may be chosen non-deterministically, but the result is still deterministic. Kahn process networks ensure deterministic results by imposing restrictions on how processes communicate so that scheduling decisions cannot be observed inside the process. All communication between processes is through first-in-first-out channels. The Kahn process network model specifically rules out processes with shared mutable state, as such a process could non-deterministically compute different results. If one process were reading from mutable state while another were writing a new value, then the reading process may get the old value or new value, depending on how the processes were scheduled. Reading from channels is blocking: processes cannot to *peek* at a channel to see whether there are waiting values, because another process might be waiting to be scheduled and about to push a new value. Channels are written to by a single process, broadcasting each value to all consumers of the channel. Only one process is ever allowed to push to a given channel: if two processes were able to push to the same channel at the same time, the scheduler would have to decide the order in which values were received.

With Kahn process networks, we can implement a process which joins sorted streams by pulling from each input channel as in the `join` combinator, and we can share streams among multiple consumers because pushed values are broadcast to each consumer. We can convert operators that use both push streams and pull streams to processes. We can also implement a process that copies values from a pull stream into a channel, and a process that copies values from a channel into a push stream.

```
data Channel a

data Network a
instance Monad Network

data Result  a
instance Applicative Result

map     :: (a → b) → Channel a
        → Network (Channel b)
join    :: (a → b → Ordering) → Channel a → Channel b
        → Network (Channel (a,b))
foldl   :: (a → b → a) → a → Channel b
        → Network (Result a)

execute :: Network (Result a) → IO a
```

Listing 2.11: Types and combinators for Kahn process networks

One version of the Kahn process network model uses bounded channels to ensure that the entire network executes in bounded memory. Kahn process networks with bounded channels still compute results deterministically, but can introduce *artificial deadlocks* in cases where the computation would succeed with a sufficiently large buffer, but the given bounds are too small. There are dynamic algorithms to identify artificial deadlocks at runtime and resolve them by increasing buffer sizes (Parks, 1995; Geilen and Basten, 2003).

Listing 2.11 shows the datatypes and type signatures of a Kahn process network implementation. We leave discussion of the implementation for Chapter 4, and for now focus solely on this simplified version of the interface. The `Channel` type denotes a communication channel between processes. The `Network` monad describes how to construct a process network; execution is deferred until after the entire network has been constructed. The `map` and `join` combinators have type signatures similar to the list versions, with lists replaced by `Channel`s and the return value inside the `Network` monad.

Because execution is deferred, the `foldl` combinator cannot return the fold result immediately, and the result is wrapped in a `Result` type. The `Result` type describes the result of executing a process network: it is a promise that the value will be available after all the processes in the network finish. The `Result` has an applicative functor instance, allowing multiple results to be combined together. The `execute` function takes a process network description containing the result promise, and executes the processes before extracting the result.

```
correlation :: Channel (Double,Double) → Network (Result Double)
regression  :: Channel (Double,Double) → Network (Result Line)

priceOverTime :: Channel Record → Network (Result (Line,Double)
priceOverTime stock = do
  timeprices ← map (λr → (daysSinceEpoch (time r), price r)) stock
  r          ← regression  timeprices
  c          ← correlation timeprices
  return ((,) <$> r <*> c)

priceOverMarket :: Channel Record → Channel Record → Network (Result (Line,Double))
priceOverMarket stock index = do
  joined ← join (λs i   → time s `compare` time i) stock index
  prices ← map  (λ(s,i) → (price s, price i))      joined
  r      ← regression  prices
  c      ← correlation prices
  return ((,) <$> r <*> c)

priceAnalyses :: Channel Record → Channel Record
              → Network (Result ((Line,Double),(Line,Double)))
priceAnalysis stock index = do
  pot ← priceOverTime   stock
  pom ← priceOverMarket stock index
  return ((,) <$> pot <*> pom)
```

Listing 2.12: Implementation of `priceAnalyses` queries as a Kahn process network

Listing 2.12 shows the `priceAnalyses` queries implemented as a Kahn process network. There are some differences from the list version: the process network is constructed inside the `Network` monad and the results are paired together using `Result` applicative functor instance. Converting the implementation from the list form to the process network form is almost purely syntactic, in contrast to the polarity analysis required for polarised streams.

Concurrent process networks have the desired high-level semantics for executing concurrent queries, but they do not provide the ideal execution strategy. In Section 2.2.1, we saw that communication between pull stream combinators involves allocating `Maybe` values, which can sometimes be removed by general purpose compiler optimisations (and sometimes not). Communication between processes requires more overhead than allocating `Maybe` values, and is not removed by general purpose optimisations. To send a value from one process to another, the sending process may need to lock the communication channel to ensure that it has exclusive access to the channel, before copying the value into a buffer where it can be read by the other process. Concurrent process network implementations amortise the cost of communication by *chunking* messages together: instead of sending many messages with one value in each, chunked communication sends one message containing an array of values. Chunking reduces the cost of sending messages, but increases memory and cache pressure. Chunk size determines how many communications are saved, so larger chunks mean less communication overhead. However, larger chunks also mean that each chunk array requires more memory and is thus less likely to fit in cache. Since each channel between a pair of processes requires its own chunk, larger process networks have more chunks in memory at the same time. The optimal chunk size is a trade-off between communication overhead and memory usage, which is usually found by experimentation.

From a functional programming perspective, small, fine-grained processes like those used in our `priceAnalysis` example are desirable because they allow us to write a process to implement each combinator and compose them together. From an execution perspective, however, when fine-grained processes perform more communication than computational work, the overall performance is dominated by synchronisation and scheduling overheads (Chen et al., 1990). We can reduce the amount of communication by fusing multiple connected processes together into one larger process. The fused process performs the task of multiple individual processes, but communicates by local variables instead of channels. In Chapter 4 we describe an algorithm to fuse processes together to reduce overhead. For `priceAnalysis`, our algorithm can automatically fuse all the processes together into a single processes. A single process executes sequentially, so fusing the entire network into a single process re-

moves any potential speedup from task parallelism, but in our benchmarks in Chapter 5, the sequential version is faster than the concurrent version even with several processors. Often, a well-optimised sequential implementation of a program will consume significantly less power and cost less to run than a parallel implementation (McSherry et al., 2015).

2.6 SUMMARY

We have seen the relative advantages of various streaming models. Pull streams support operators with multiple inputs, and can take advantage of an optimising compiler to reduce overhead. Push streams support multiple concurrent queries, and are written back-to-front with explicit duplication for sharing streams. Polarised streams support multiple inputs and multiple queries, require polarity analysis of the entire dependency graph, and are written partially back-to-front and partially front-to-back. Kahn process networks support multiple inputs and multiple queries, and concurrent execution involves communication overhead.

In the next chapter we will see Icicle, a language for specifying push stream queries. Icicle queries are written front-to-back, and streams can be shared without requiring explicit duplication. Queries are compiled to folds over push streams which can be executed concurrently. After looking at Icicle, we shall see how Kahn process networks can be executed efficiently by fusing processes together (Chapter 4).

CHAPTER 3

ICICLE, A LANGUAGE FOR PUSH QUERIES

This chapter presents Icicle, a domain-specific language for writing queries as push streams. This work was first published as Robinson and Lippmeier (2016), and was performed in collaboration with a machine-learning company called Ambiata. At Ambiata, we perform feature generation for machine-learning applications by executing many thousands of simple queries over terabytes worth of compressed data.[1] For such applications, we must automatically fuse these separate queries and be sure that the result can be executed in a single pass over the input. We also ingest tens of gigabytes of new data per day, and must incrementally update existing features without recomputing them all from scratch. Our feature generation process is executed in parallel on hundreds of nodes on a cloud-based system, and if we performed neither fusion nor incremental update then the cost of the computation would begin to exceed the salaries of the developers.

The contributions of this chapter are:

- We motivate the use of Icicle by extending the previous "gold panning" queries (Section 3.1);
- We present Icicle, a domain-specific language that guarantees that any set of queries on a shared input table can be fused, and allows the query results to be updated as new data is received (Section 3.2.5);
- We present a fold-based intermediate language, which allows the query fusion transformation to be a simple matter of appending two intermediate programs, and exposes opportunities for common subexpression elimination (Section 3.3);
- We present benchmarks of Icicle compiled code running in production (Section 3.4).

Our implementation is available at `https://github.com/amosr/icicle`. This implementation has been running in production at Ambiata for over two years.

[1] In 2018, this was a lot of data.

3.1 GOLD PANNING WITH ICICLE

For the following example queries, we extend the daily stock price records from Section 2.1 to contain the open price and the close price for the day. The queries we write process a single input stream of prices for a particular company, and Icicle uses push streams to ensure that all queries over the same input stream can be fused together. Icicle cannot express the `priceOverMarket` example, which uses multiple input streams, but we will return to this limitation in Chapter 4.

Suppose we want to compute the number of days where the open price exceeded the close price, and vice versa. We also want to compute the mean of the open price for days in which the open price exceeded the close price. In Icicle, we write the three queries as follows:

```
table stocks { open : Int, close : Int }
query
  more = filter open > close of count;
  less = filter open < close of count;
  mean = filter open > close of sum open / count;
```

In the above code, (`open > close`) and (`close < open`) are filter predicates, and `count` counts how many times the predicate is true. The input table, `stocks`, defines the open and close prices as `Int`s. In Icicle, input tables have an implicit time field and the input stream is sorted chronologically.

Listing 3.1 shows the same three queries implemented using the push streams from Section 2.3. Despite the syntactic differences, the two programs have roughly the same structure in terms of the three queries. The three instances of `count` are constructed as monadic `IO` operations, because each count uses a separate mutable reference. The applicative functor syntax is used to divide the sum by the count in the `mean` query, because the division is performed on the result of the push stream. Icicle does not use the applicative syntax, as it uses a modal type system to infer which computations are performed on the result of the stream, as described in Section 3.2.6.

In this example, both `more` and `mean` compute the count of elements that match the same filter predicate. When the same value would be computed by multiple queries, we would instead like to compute the value only once and share the result among all the queries that use it. Common subexpression elimination (CSE) removes some duplicate computations but, as its name suggests, it is limited to structural subexpressions (Chitil, 1997b). Neither of the filtered counts is a subexpression of the other, so common subexpression elimination will not remove

```
data Record = Record
 { time :: Time, open :: Int, close :: Int }

queries :: IO (Push Record (Int,Int,Int))
queries = do
  more_count ← count
  let more = filter (λr → open r > close r) more_count

  less_count ← count
  let less = filter (λr → open r < close r) less_count

  mean_sum   ← foldl (+) 0
  mean_count ← count
  let mean = filter (λr → open r > close r)
                    (div <$> contramap open mean_sum <*> mean_count)

  return ((,,) <$> more <*> less <*> mean)
```

Listing 3.1: Push implementation of queries

the duplicate computation. In Icicle, we remove this duplicate work by first converting queries to an intermediate language, described in Section 3.3. This intermediate language decomposes the query into individual folds, exposing the opportunities for common subexpression elimination.

If we were using an existing database implementation, we could convert all three queries to a single query in a back-end language like SQL, but doing so by hand is tedious and error prone. As the three queries use different filter predicates, we cannot use a single `SELECT` statement with a `WHERE` expression to implement the filter. We must instead lift each predicate to an expression-level conditional and compute the count by summing the conditional:

```
SELECT SUM(IF(open > close, 1,    0))
     , SUM(IF(open < close, 1,    0))
     , SUM(IF(open > close, open, 0))
     / SUM(IF(open > close, 1,    0))
FROM stocks;
```

Joint queries such as the stocks example can be evaluated in a streaming, incremental fashion, which allows the result to be updated as we receive new data (Arasu et al., 2003). As a counter-example, suppose we have a table with two fields `key` and `value`, and we wish to

find the mean of values whose key matches the last one in the table. We might try something like the following:

```
table kvs { key : Date; value : Int }
query meanOfLatest
 = let k = last key in
   filter (key == k) of mean value;
```

Unfortunately, although the *result* we desire is computable, the *algorithm* implied by the above query cannot be evaluated incrementally. When we are streaming through the table we always have access to the last key in the stream, but finding the rows that match this key requires streaming the table again from the start. We need a better solution.

Icicle is related to stream processing languages such as Lucy (Mandel et al., 2010) and StreamIt (Thies et al., 2002), except we forgo the need for clock and deadlock analysis. Icicle is also related to work on continuous queries (Arasu et al., 2003), where query results are updated as rows are inserted into the source table, except we can also compute arbitrary reductions and do not need to handle deleted source rows. We discuss these points in more detail in Section 3.5.

3.2 ELEMENTS AND AGGREGATES

To allow incremental computation, all Icicle queries must execute in a single pass over the input stream. Sadly, not all queries *can* be executed in a single pass: the key examples are queries that require random access indexing, or otherwise need to access data in an order different to what the stream provides. However, as we saw in the previous section, although a particular *algorithm* may be impossible to evaluate in a streaming fashion, the desired *value* may well be computable, if only we had a different algorithm. Here is the unstreamable example from the previous section again:

```
table kvs { key : Date; value : Int }
query meanOfLatest
 = let k = last key in
   filter (key == k) of mean value;
```

The problem is that the value of `last key` is only available once we have reached the end of the stream, but `filter` needs this value to process the very first element in the same stream. We distinguish between these two access patterns by giving them different names: we say

that the expression (`last key`) is an *aggregate*, because to compute it we must have consumed the *entire stream*, whereas the filter predicate is an *element*-wise computation because it only needs access to the current element in the stream.

The trick to compute our average in a streaming fashion is to recognise that `filter` selects a particular subset of values from the input, but the value computed from this subset depends only on the values in that subset, and no other information. Instead of computing the mean of a single subset whose identity is only known at the end of the stream, we can instead compute the mean of *all possible subsets*, and return the required one once we know what that is:

```
table kvs { key : Date; value : Int }
query meanOfLatest
 = let k    = last  key in
   let avgs = group key of mean value in
   lookup k avgs
```

Here we use the `group` construct to assign key-value pairs to groups as we obtain them, and compute the running mean of the values of each group. The `avgs` value becomes a map of group keys to their running means. Once we reach the end of the stream we will have access to the last key and can retrieve the final result. Evaluation and typing rules are defined in Section 3.2.5, while the user functions `last` and `mean` are defined in Section 3.3.

3.2.1 *The stage restriction*

To ensure that Icicle queries can be evaluated in a single pass, we use a modal type system inspired by staged computation (Davies and Pfenning, 2001). We use two modalities, `Element` and `Aggregate`. Values of type `Element` τ are taken from the input stream on a per-element basis, whereas values of type `Aggregate` τ are available only once the entire stream has been consumed. In the expression (`filter (key == k) of mean value`), the variable `key` has type `Element Date` while `k` has type `Aggregate Date`. Attempting to compile the unstreamable query in Icicle will produce a type error complaining that elements cannot be compared with aggregates.

The types of pure values, such as constants, are automatically promoted to the required modality. For example, if we have the expression (`open == 1`), and the type-checking environment asserts that the variable `open` has type `Element Int`, then the constant `1` is automatically promoted from type `Int` to type `Element Int`.

3.2.2 *Finite streams and synchronous data flow*

In contrast to synchronous data flow languages such as Lustre (Halbwachs et al., 1991), the streams processed by Icicle are conceptually finite in length. Icicle is fundamentally a query language, that queries finite tables of data held in a non-volatile store, but does so in a streaming manner. Lustre operates on conceptually infinite streams, such as those found in real-time control systems (like to fly aeroplanes). In Icicle, the "last" element in a stream is the last one that appears in the table on disk. In Lustre, the "last" element in a stream is the one that was most recently received.

If we took the unstreamable query from Section 3.2 and converted it to Lustre syntax, then the resulting program would execute, but the filter predicate would compare the last key with the most recent key from the stream, which is the key itself. The filter predicate would always be true, and the query would return the mean of the entire stream. Applying the Icicle type system to our queries imposes the natural stage restriction associated with finite streams, so there are distinct "during" (element) and "after" (aggregate) stages.

3.2.3 *Incremental update*

Suppose we query a large table and record the result. Tomorrow morning, we receive more data and add it to the table. We would like to update the result without needing to process all data from the start of the table. We can perform this incremental update by remembering the values of all intermediate aggregates that were computed in the query, and updating them as new data arrives. In the streamable version of the `meanOfLatest` example from Section 3.2, these aggregates are `k` and `avgs`.

We also provide impure contextual information to the query, such as the current date, by assigning it an aggregate type. As element-wise computations cannot depend on aggregate computations, we ensure that reused parts of an incremental computation are the same regardless of which day they are executed.

3.2.4 *Bounded buffer restriction*

Icicle queries process tables of arbitrary size that may not fit in memory. As with other streaming models, each query must execute without requiring buffer space proportional to the size

of the input. As a counterexample, here is a simple list function that cannot be executed in a streaming manner without reserving a buffer of the same size as the input:

```
unbounded :: [Int] → [(Int,Int)]
unbounded xs = zip (filter (> 0) xs) (filter (< 0) xs)
```

This function takes an input list `xs`, and pairs the elements that are greater than zero with those that are less than zero. If we try to convert this computation to a single-pass streaming implementation, it requires an unbounded buffer: if the input stream contains n positive values followed by n negative values, then all positive values must be buffered until we reach the negative ones, which allow output to be produced.

In Icicle, queries that would require unbounded buffering are statically outlawed by the type system, with one major caveat that we will discuss in a moment. Because Icicle is based on the push streams described in Section 2.3, the stream being processed (such as `xs` above) is implicit in each query. Constructs such as `filter` and `fold` do not take the name of the input stream as an argument, but instead operate on the stream defined in the context. Icicle language constructs describe *how elements from the stream should be aggregated*, but the order in which those elements are aggregated is implicit, rather than being definable by the body of the query. In the expression (`filter p of mean value`), the term (`mean value`) is applied to stream values which satisfy the predicate `p`, but the values to consider are supplied by the context.

Finally, our major caveat is that the `group` construct we used in Section 3.2 uses space proportional to the number of distinct *keys* in the input stream. For our applications, the keys are commonly company names, customer names, and days of the year. Our production system knows that these types are bounded in size, and that maps from keys to values will fit easily in memory. Attempting to group by values of a type with an unbounded number of members, such as a `Real` or `String`, results in a compile-time warning.

3.2.5 *Source language*

The grammar for Icicle is given in Figure 3.1. Value types (T) include numbers, booleans and maps; some types such as `Real` and `String` are omitted. Modal types (M) include the pure value types, and modalities associated with a value type. Function types (F) include functions with any number of modal type arguments to a modal return type. As Icicle is a first-order language, function types are not value types.

$$
\begin{array}{lll}
T & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{Map}\ T\ T \mid (T \times T) \\
\\
M & ::= & T \mid \texttt{Element}\ T \mid \texttt{Aggregate}\ T \\
\\
F & ::= & \overline{(M)} \rightarrow M \\
\\
\mathit{Table} & ::= & \texttt{table}\ x\ \{\ \overline{(x\ :\ T;)}\ \} \\
\\
\mathit{Exp}, e & ::= & x \mid V \mid \mathit{Prim}\ \overline{\mathit{Exp}} \mid x\ \overline{\mathit{Exp}} \\
& \mid & \texttt{let}\ \ x\ = \mathit{Exp}\ \ \texttt{in}\ \mathit{Exp} \\
& \mid & \texttt{fold}\ \ x\ = \mathit{Exp}\ \texttt{then}\ \mathit{Exp} \\
& \mid & \texttt{filter}\ \ \ \mathit{Exp}\ \ \texttt{of}\ \mathit{Exp} \\
& \mid & \texttt{group}\ \ \ \mathit{Exp}\ \ \texttt{of}\ \mathit{Exp} \\
\\
\mathit{Prim}, p & ::= & (\texttt{+}) \mid (\texttt{-}) \mid (\texttt{*}) \mid (\texttt{/}) \mid (\texttt{==}) \mid (\texttt{/=}) \mid (\texttt{<}) \mid (\texttt{>}) \mid (\texttt{,}) \\
& \mid & \texttt{lookup} \mid \texttt{fst} \mid \texttt{snd} \\
\\
V, v & ::= & \mathbb{N} \mid \mathbb{B} \mid \{V \Rightarrow V\} \mid (V \times V) \\
\\
\mathit{Def} & ::= & \texttt{function}\ f\ \overline{(x\ :\ M)}\ =\ \mathit{Exp} \\
& \mid & \texttt{query}\ x\ =\ \mathit{Exp} \\
\\
\mathit{Top} & ::= & \mathit{Table};\ \overline{\mathit{Def}};
\end{array}
$$

Figure 3.1: Icicle grammar

Table definitions (*Table*) define a table name and the names and types of columns.

Expressions (*Exp*) include variable names, constants, applications of primitives and functions. The `fold` construct defines the name of an accumulator, the expression for the initial value, and the expression used to update the accumulator for each element of the stream. The `filter` construct defines a predicate and an expression to accumulate values for which the predicate is true. The `group` construct defines an expression used to determine the key for each element of the stream, and an expression to accumulate the values that share a common key.

Grammar *Prim* defines the primitive operators. Grammar *V* defines values. Grammar *Def* contains both function and query definitions. Grammar *Top* is the top-level program, which specifies a table, the set of function bindings, and the set of queries. All queries in a top-level program process the same input table.

3.2.6 *Type system*

The typing rules for Icicle are given in Figure 3.2. The judgment form ($\Gamma \vdash e : M$) associates an expression e with its type M under context Γ. The judgment form ($p :_P F$) associates a primitive with its function type F. The judgment form ($F \bullet \overline{M} : M$) is used to lift function application to modal types: a function type F applied to a list of modal argument types $\overline{M}$ produces a result type and matching mode M. The judgment form ($\Gamma \vdash Def \dashv \Gamma$) takes an input environment Γ and function or query, and produces an environment containing the function or query name and its type. Finally, the judgment form ($\vdash Top \dashv \Gamma$) takes a top-level definition with a table, functions and queries, and produces a context containing the types of all the definitions.

Rules (TcNat), (TcBool), (TcMap) and (TcPair) assign types to literal values. Rule (TcVar) performs variable lookup in the context. Rule (TcBox) performs the promotion mentioned earlier, allowing a pure expression to be implicitly treated as an `Element` or `Aggregate` type.

Rules (TcPrimApp) and (TcFunApp) produce the type of a primitive or function applied to its arguments, using the auxiliary judgment forms for application. Rule (TcLet) is standard.

In rule (TcFold), the initial value has value type T. A binding for the fold accumulator is added to the context of e_k with type (`Element` T), and the result of the overall fold has type (`Aggregate` T).

Rule (TcFilter) requires the first argument of a `filter` to have type (`Element Bool`), denoting a stream of predicate flags. The second argument must have modality `Aggregate`, denoting a

$$\boxed{\Gamma \vdash e : M}$$

$$\frac{}{\Gamma \vdash \mathbb{N} : \texttt{Int}}\ (\text{TcNat}) \qquad \frac{}{\Gamma \vdash \mathbb{B} : \texttt{Bool}}\ (\text{TcBool}) \qquad \frac{\{\Gamma \vdash v_i : T\} \quad \{\Gamma \vdash v_i' : T'\}}{\Gamma \vdash \{v_i \Rightarrow v_i'\} : \texttt{Map}\ T\ T'}\ (\text{TcMap})$$

$$\frac{\Gamma \vdash v : T \quad \Gamma \vdash v' : T'}{\Gamma \vdash v \times v' : T \times T'}\ (\text{TcPair}) \qquad \frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}\ (\text{TcVar})$$

$$\frac{\Gamma \vdash e : T \quad m \in \{\texttt{Element}, \texttt{Aggregate}\}}{\Gamma \vdash e : m\ T}\ (\text{TcBox})$$

$$\frac{p :_P F \quad \{\Gamma \vdash e_i : M_i\} \quad F \bullet \{M_i\} : M'}{\Gamma \vdash p\ \{e_i\} : M'}\ (\text{TcPrimApp})$$

$$\frac{(x : F) \in \Gamma \quad \{\Gamma \vdash e_i : M_i\} \quad F \bullet \{M_i\} : M'}{\Gamma \vdash x\ \{e_i\} : M'}\ (\text{TcFunApp}) \qquad \frac{\Gamma \vdash e : M \quad \Gamma, x : M \vdash e' : M'}{\Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ e' : M'}\ (\text{TcLet})$$

$$\frac{\Gamma \vdash e_z : T \quad \Gamma, x : \texttt{Element}\ T \vdash e_k : \texttt{Element}\ T}{\Gamma \vdash \texttt{fold}\ x = e_z\ \texttt{then}\ e_k : \texttt{Aggregate}\ T}\ (\text{TcFold})$$

$$\frac{\Gamma \vdash e : \texttt{Element Bool} \quad \Gamma \vdash e' : \texttt{Aggregate}\ T}{\Gamma \vdash \texttt{filter}\ e\ \texttt{of}\ e' : \texttt{Aggregate}\ T}\ (\text{TcFilter})$$

$$\frac{\Gamma \vdash e : \texttt{Element}\ T \quad \Gamma \vdash e' : \texttt{Aggregate}\ T'}{\Gamma \vdash \texttt{group}\ e\ \texttt{of}\ e' : \texttt{Aggregate}\ (\texttt{Map}\ T\ T')}\ (\text{TcGroup})$$

$$\boxed{p :_P F}$$

$$\frac{p \in \{+, -, *, /\}}{p :_P (\texttt{Int}, \texttt{Int}) \to \texttt{Int}}\ (\text{PrimArith}) \qquad \frac{p \in \{==, /=, <, >\}}{p :_P (\texttt{Int}, \texttt{Int}) \to \texttt{Bool}}\ (\text{PrimRel})$$

$$\frac{}{(,) :_P (T, T') \to (T \times T')}\ (\text{PrimTuple}) \qquad \frac{}{\texttt{lookup} :_P (\texttt{Map}\ T\ T', T) \to T'}\ (\text{PrimLookup})$$

$$\frac{}{\texttt{fst} :_P (T \times T') \to T}\ (\text{PrimFst}) \qquad \frac{}{\texttt{snd} :_P (T \times T') \to T'}\ (\text{PrimSnd})$$

$$\boxed{F \bullet \overline{M} : M}$$

$$\frac{}{(\{M_i\} \to M') \bullet \{M_i\} : M'}\ (\text{AppArgs}) \qquad \frac{}{(\{T_i\} \to T') \bullet \{m\ T_i\} : m\ T'}\ (\text{AppRebox})$$

$$\boxed{\Gamma \vdash Def \dashv \Gamma}$$

$$\frac{\Gamma \cup \{x_i : M_i\} \vdash e : M' \quad F = \{M_i\} \to M'}{\Gamma \vdash \texttt{function}\ x\ \{x_i : M_i\} = e \dashv \Gamma, x : F}\ (\text{CheckFun})$$

$$\frac{\Gamma \vdash e : \texttt{Aggregate}\ T}{\Gamma \vdash \texttt{query}\ x = e \dashv \Gamma, x : \texttt{Aggregate}\ T}\ (\text{CheckQuery})$$

$$\boxed{\vdash Top \dashv \Gamma}$$

$$\frac{\Gamma_0 = \{x_i : \texttt{Element}\ T_i\} \quad \{\Gamma_{j-1} \vdash d_j \dashv \Gamma_j\}}{\vdash \texttt{table}\ x\ \{x_i : T_i\}; \{d_j\} \dashv \Gamma_j}\ (\text{CheckTop})$$

Figure 3.2: Types of expressions

fold to perform over the filtered elements. The result is also an `Aggregate` of the same type as the fold. By restricting `filter` to only perform folds, we ban `filter` from returning a stream of elements of a different length, and side-step the issue of clock analysis. We discuss clock types further in Section 3.5.

Rule (TcGroup) performs a similar nested aggregation to `filter`.

Rules (PrimArith), (PrimRel), (PrimTuple), (PrimLookup), (PrimFst) and (PrimSnd) assign types to primitives.

Rule (AppArgs) produces the type of a function or primitive applied to its arguments. Rule (AppRebox) is used when the arguments have modal type m — applying a pure function to arguments of mode m produces a result of the same mode.

Rule (CheckFun) builds the type of a user defined function, returning it as an element of the output context. Rule (CheckQuery) is similar, noting that all queries return values of `Aggregate` type. Finally, rule (CheckTop) checks a whole top-level program.

3.2.7 *Evaluation*

We now give a denotational evaluation semantics for Icicle queries. For the evaluation semantics, we introduce an auxiliary grammar for describing *stream values* and heaps. In the source language, all streams are the same length and rate as the input stream, to ensure that elements from different streams can always be pairwise joined. To maintain this invariant, size-changing operations such as `filter` perform folds rather than returning differently-sized output streams. Introducing literal stream values and representing them as a list of values would invalidate this invariant, because the length of the input stream is not statically known. Instead, in the evaluation semantics, we represent `Element` stream values as meta-level stream transformers, which transform the input element to an output element. Likewise, we represent `Aggregate` values as meta-level folds. The result of evaluating a query will also be a meta-level fold, into which the values from the input stream are pushed. We introduce the definition of stream values in the evaluation semantics only, which forces us to use a heap-based semantics instead of a substitution-based semantics.

The auxiliary grammar and evaluation rules for Icicle are given in Figure 3.3. Grammar N defines the modes of evaluation, including pure computation. Grammar Σ defines a heap containing stream values, where each assignment has an associated evaluation mode. Grammar V' defines the stream values that can be produced by evaluation, depending on the mode:

- `Pure` computation results are a single value;

- `Element` computation results are stream transformers, which are represented by meta-functions that take a value of the input stream element and produce an output stream element value; and
- `Aggregate` computation results consist of an initial state, an update meta-function to be applied to each stream element and current state, and an eject meta-function to be applied to the final state to produce the final result value.

In the grammar V', we write $(\overset{\bullet}{\rightarrow})$ to highlight that the objects in those positions are meta-functions, rather than abstract syntax. To actually process data from the input table, we will need to apply the produced meta-functions to the data.

The judgment form $(N \mid \Sigma \vdash e \Downarrow V')$ defines a big-step evaluation relation: under evaluation mode N with heap Σ, expression e evaluates to result V'. The evaluation mode N controls whether pure values should be promoted to element (stream) or aggregate (fold) results. We assume that all functions have been inlined into the expression before evaluation.

Rule (EVal) applies when the expression is a constant value. Rule (EVar) performs variable lookup in the heap, and requires the evaluation mode to be the same as the mode of the variable. Rule (ELet) evaluates the bound expression under the given mode, and inserts the binding into the heap.

Rules (EBoxStream) and (EBoxFold) lift pure values to stream results and aggregate results respectively. To lift a pure value to a stream result, we produce a meta-function that always returns the value. To lift a pure value to an aggregate result, we set the update meta-function to return a dummy value, and have the eject meta-function return the value of interest.

Rules (EPrimValue), (EPrimStream) and (EPrimFold) apply primitive operators to pure values, streams and aggregations respectively. In (EPrimValue), all the argument expressions are bound in the sequence e using the sequence comprehension syntax $\{e_i\}$. Each argument expression e_i is evaluated to a corresponding pure value v_i, to which the primitive operator is then applied.

Rule (EPrimStream) is similar to (EPrimValue), except the result is a new stream transformer that applies the primitive to each of the elements gained from the input streams.

In (EPrimFold), each argument expression is evaluated to a fold. Each argument's fold has its own initial fold state (z), update function (k) and eject function (j). The result fold's initial state is the tuple of all arguments' initial states ($\prod_i z_i$). The result fold's update function applies each argument's update functions to the input stream element (s) and the corresponding accumulator state (v_i). The result fold's eject function performs all arguments' ejects and applies the primitive operator to the final result of all argument folds.

$$N ::= \texttt{Pure} \mid \texttt{Element} \mid \texttt{Aggregate} \quad \Sigma ::= \cdot \mid \Sigma, x =_N V'$$

$$V' ::= \texttt{Value}\ V \mid \texttt{Stream}\ (V \xrightarrow{\bullet} V) \mid \texttt{Fold}\ V\ (V \xrightarrow{\bullet} V \xrightarrow{\bullet} V)\ (V \xrightarrow{\bullet} V)$$

$$\boxed{N \mid \Sigma \vdash e \Downarrow V'}$$

$$\frac{}{\texttt{Pure} \mid \Sigma \vdash V \Downarrow \texttt{Value}\ V}\ (\text{EVal}) \qquad \frac{x =_n V' \in \Sigma}{n \mid \Sigma \vdash x \Downarrow V'}\ (\text{EVar}) \qquad \frac{n' \mid \Sigma \vdash e \Downarrow v \quad n \mid \Sigma, x =_{n'} v \vdash e' \Downarrow v'}{n \mid \Sigma \vdash \texttt{let}\ (x : n'\ \tau') = e\ \texttt{in}\ e' \Downarrow v'}\ (\text{ELet})$$

$$\frac{\texttt{Pure} \mid \Sigma \vdash e \Downarrow \texttt{Value}\ v}{\texttt{Element} \mid \Sigma \vdash e \Downarrow \texttt{Stream}\ (\lambda s.\ v)}\ (\text{EBoxStream}) \qquad \frac{\texttt{Pure} \mid \Sigma \vdash e \Downarrow \texttt{Value}\ v}{\texttt{Aggregate} \mid \Sigma \vdash e \Downarrow \texttt{Fold}\ ()\ (\lambda s\ ().\ ())\ (\lambda().\ v)}\ (\text{EBoxFold})$$

$$\frac{\{\texttt{Pure} \mid \Sigma \vdash e_i \Downarrow \texttt{Value}\ v_i\}}{\texttt{Pure} \mid \Sigma \vdash p\ \{e_i\} \Downarrow \texttt{Value}\ (p\ \{v_i\})}\ (\text{EPrimValue}) \qquad \frac{\{\texttt{Element} \mid \Sigma \vdash e_i \Downarrow \texttt{Stream}\ v_i\}}{\texttt{Element} \mid \Sigma \vdash p\ \{e_i\} \Downarrow \texttt{Stream}\ (\lambda s.\ p\ \{v_i\ s\})}\ (\text{EPrimStream})$$

$$\frac{\{\texttt{Aggregate} \mid \Sigma \vdash e_i \Downarrow \texttt{Fold}\ z_i\ k_i\ j_i\}}{\texttt{Aggregate} \mid \Sigma \vdash p\ \{e_i\} \Downarrow \texttt{Fold}\ (\prod_i z_i)\ (\lambda s\ v.\ \prod_i (k_i\ s\ v_i))\ (\lambda(\prod_i v_i).\ p\ \{j_i\ v_i\})}\ (\text{EPrimFold})$$

$$\frac{\texttt{Element} \mid \Sigma \vdash e \Downarrow \texttt{Stream}\ f \quad \texttt{Aggregate} \mid \Sigma \vdash e' \Downarrow \texttt{Fold}\ z\ k\ j}{\texttt{Aggregate} \mid \Sigma \vdash \texttt{filter}\ e\ \texttt{of}\ e' \Downarrow \texttt{Fold}\ z\ (\lambda s\ v.\ \texttt{if}\ f\ s\ \texttt{then}\ k\ s\ v\ \texttt{else}\ v)\ j}\ (\text{EFilter})$$

$$\frac{\texttt{Element} \mid \Sigma \vdash e \Downarrow \texttt{Stream}\ f \quad \texttt{Aggregate} \mid \Sigma \vdash e' \Downarrow \texttt{Fold}\ z\ k\ j}{\texttt{Aggregate} \mid \Sigma \vdash \texttt{group}\ e\ \texttt{of}\ e' \Downarrow \texttt{Fold}\ \{\_ \Rightarrow z\}\ k'\ j'}\ (\text{EGroup})$$
$$\text{where } k' = \lambda s\ m.\ (f\ s \Rightarrow k\ s\ (m[f\ s])) \cup m$$
$$j' = \lambda m.\ \{k_i \Rightarrow j\ v_i \mid k_i \Rightarrow v_i \in m\}$$

$$\frac{\texttt{Pure} \mid \Sigma \vdash z \Downarrow \texttt{Value}\ z' \quad \texttt{Element} \mid \Sigma' \vdash k \Downarrow \texttt{Stream}\ k'}{\texttt{Aggregate} \mid \Sigma \vdash \texttt{fold}\ x = z\ \texttt{then}\ k \Downarrow \texttt{Fold}\ z'\ (\lambda s\ v.\ k'\ (v, s))\ (\lambda v.\ v)}\ (\text{EFold})$$
$$\text{where } \Sigma' = (x = \texttt{Stream}\ \texttt{fst}),$$
$$\{x_i = \texttt{Stream}\ (f_i \cdot \texttt{snd}) \mid x_i = \texttt{Stream}\ f_i \in \Sigma\},$$
$$\{x_i = \texttt{Fold}\ z_i\ (k_i \cdot \texttt{snd})\ j_i \mid x_i = \texttt{Fold}\ z_i\ k_i\ j_i \in \Sigma\},$$
$$\{x_i = \texttt{Value}\ v_i \mid x_i = \texttt{Value}\ v_i \in \Sigma\}$$

$$\boxed{\{x \Rightarrow \overline{V}\} \mid e \Downarrow V}$$

$$\frac{\texttt{Aggregate} \mid \{x_i = \texttt{Stream}\ (\texttt{fst} \cdot \texttt{snd}^i) \mid x_i \Rightarrow v_i \in t\} \vdash e \Downarrow \texttt{Fold}\ z\ k\ j}{t \mid e \Downarrow j\ (\mathit{fold}\ k\ z\ \{v_0 \times \cdots \times v_i \times () \mid x_i \Rightarrow v_i \in t\})}\ (\text{ETable})$$

Figure 3.3: Evaluation rules and auxiliary grammar

Rule (EFilter) first evaluates the predicate e to a stream transformer f, and the body e' to an aggregation. The result is a new aggregation where the update function applies the predicate stream transformer f to the input element s to yield a boolean flag which specifies whether the current aggregation state should be updated.

Rule (EGroup) is similar to (EFilter), except that the stream transformer f produces group keys rather than boolean flags, and we maintain a finite map of aggregation states for each key. In the result aggregation, the update function (k') updates the appropriate accumulator in the map, and the eject function (j') applies the original eject function to every accumulator in the map.

Rule (EFold) introduces a new accumulator, which is visible in the context of the body k. Evaluating the body k produces a body stream transformer k', whose job is to update this new accumulator each time it is applied. This stream transformer takes as input a tuple containing the current accumulator value and the input stream element, and returns the updated accumulator value. We introduce a heap binding for the new accumulator, which extracts the accumulator value from the first element of the input tuple. When the k' stream transformer uses any other stream transformer bindings from the heap, it will pass the tuple containing the accumulator value and the stream element. The existing stream transformer bindings from the heap are only expecting to receive the stream element, so we modify the heap bindings to extract the stream element before applying the transformer. In the conclusion of (EFold), we return a fold result. The fold's update function passes the stream transformer a tuple (v, s), where v is the accumulator value and s is the input element of the stream received from the context of the overall `fold` expression.

The judgment form $(t \mid e \Downarrow V)$ evaluates an expression over a table input: on input table t, aggregate expression e evaluates to value V. The input table t is a map from column name to a list of all the values for that column. Rule (ETable) creates an initial heap where each column name x_i is bound to an expression which projects out the appropriate element from a single row in the input table. Evaluating the expression e produces an aggregation result where the update function k accepts each row from the table and updates all the accumulators defined by e. The actual computation is driven by the *fold* meta-function.

3.3 INTERMEDIATE LANGUAGE

To execute Icicle queries over large datasets, we first convert the queries to an intermediate language that is similar to a physical query plan for a database system. We convert each source

$$
\begin{array}{lll}
\mathit{PlanX} & ::= & x \mid V \mid \mathit{PlanP}\ \overline{\mathit{PlanX}} \mid \lambda x.\ \mathit{PlanX} \\
\mathit{PlanP} & ::= & \mathit{Prim} \mid \texttt{mapUpdate} \mid \texttt{mapEmpty} \mid \texttt{mapMap} \mid \texttt{mapZip} \\
\mathit{PlanF} & ::= & \texttt{fold}\ \ x\ :\ T\ =\ \mathit{PlanX}\ \texttt{then}\ \mathit{PlanX}; \\
 & \mid & \texttt{filter}\ \ \mathit{PlanX}\ \ \{\ \overline{\mathit{PlanF}}\ \} \\
 & \mid & \texttt{group}\ \ \mathit{PlanX}\ \ \{\ \overline{\mathit{PlanF}}\ \}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{Plan} & ::= & \texttt{plan}\ x\ \ \{\ \overline{x\ :\ T;}\ \} \\
 & & \texttt{before}\ \ \{\ \overline{x\ :\ T\ =\ \mathit{PlanX};}\ \} \\
 & & \texttt{folds}\ \ \{\ \overline{\mathit{PlanF}}\ \} \\
 & & \texttt{after}\ \ \{\ \overline{x\ :\ T\ =\ \mathit{PlanX};}\ \} \\
 & & \texttt{return}\ \ \{\ \overline{x\ :\ T\ =\ x;}\ \}
\end{array}
$$

Figure 3.4: Query plan grammar

query to a query plan, then fuse together the plans for queries on the same table. Once we have the fused query plan, we then perform standard optimisations such as common subexpression elimination and partial evaluation.

The grammar for the Icicle intermediate language is given in Figure 3.4. Expressions *PlanX* include variables, values, applications of primitives and anonymous functions. Function definitions and uses are not needed in expressions here, as their definitions are inlined before converting to query plans. Anonymous functions are only allowed as arguments to primitives: they cannot be applied or bound to variables. The primitives of the source language are extended with key-value map primitives, which are used for implementing groups. Folds are defined in *PlanF* and can be nested inside a filter, in which case the accumulator of the fold is only updated when the predicate is true; the nested fold binding is available outside of the filter. Folds nested inside a group are performed separately for each key; the nested fold binds a key-value map instead of a single value. The *Plan* itself is split into a five stage *loop anatomy* (Shivers, 2005). First we have the name of the table and the names and element types of each column. The `before` stage then defines pure values which do not depend on any table data. The `folds` stage defines element computations and how they are converted to aggregate results. The `after` stage defines aggregate computations that combine multiple aggregations after the entire table has been processed. Finally, the `return` stage specifies the output values of the query; a single query will have only one output value, but the result of fusing many queries can have many outputs. When there are no bindings in a particular stage, we omit that stage completely.

Before we discuss an example query plan we first define the `count` and `sum` functions used in earlier sections. Both functions are defined as simple folds:

```
function count
 = fold c = 0 then c + 1;

function sum (e : Element Int)
 = fold s = 0 then s + e;
```

Inlining these functions into the three stocks queries from Section 3.1 yields the following set of queries:

```
table stocks { open : Int, close : Int }
query
  more = filter open > close of (fold more_c = 0 then more_c + 1);
  less = filter open < close of (fold less_c = 0 then less_c + 1);
  mean = filter open > close of
      (fold mean_s = 0 then mean_s + open)
    / (fold mean_c = 0 then mean_c + 1);
```

We convert each query to a query plan separately. When we convert the `more` query, we define the count as a fold inside a filter, and use the count binding in the `return` section to define the output of the query:

```
plan stocks { open : Int; close : Int; }
folds {
  filter open > close {
    fold c    : Int = 0 then c + 1; } }
return { more : Int = c; }
```

The `less` query follows the same structure as the `more` query, with a different predicate:

```
plan stocks { open : Int; close : Int; }
folds {
  filter open < close {
    fold c    : Int = 0 then c + 1; } }
return { less : Int = c; }
```

To convert the `mean` query, the folds for the sum and the count are both defined inside the same filter. The division is performed in the `after` section because it is an aggregate operation on the final value of the two folds:

```
plan stocks { open : Int; close : Int; }
folds {
  filter open > close {
    fold c    : Int = 0 then c + 1;
    fold s    : Int = 0 then s + open; } }
after  { sc   : Int = s / c; }
return { mean : Int = sc; }
```

To fuse the three query plans together, we freshen the names of each binding, then simply concatenate the corresponding parts of the anatomy. The single-pass restriction on queries makes the fusion process so simple, because it ensures that there are no fusion-preventing dependencies between any two query plans. We discuss fusion-preventing dependencies further in Chapter 7. After concatenating the plans, we merge the filter blocks for `more` and `mean`, as both use the same predicate. When each query was expressed separately, we were free to transform each individual query without affecting the others. Now the code that implements each query is interspersed, but the stages are expressed separately, so we are free to rearrange the bindings in each stage without affecting the other stages. The result of fusing the three query plans is as follows:

```
plan stocks { open : Int; close : Int; }
folds {
  filter open > close {
    fold more_c  : Int = 0 then more_c + 1;
    fold mean_c  : Int = 0 then mean_c + 1;
    fold mean_s  : Int = 0 then mean_s + open; }
  filter open < close {
    fold less_c  : Int = 0 then less_c + 1; } }
after  { mean_sc : Int = mean_s / mean_c }
return { more    : Int = more_c;
         less    : Int = less_c;
         mean    : Int = mean_sc; }
```

We can now use common subexpression elimination to remove the duplicate count, `mean_c`, as its binding is alpha-equivalent to the binding for `more_c`. In the after section, the reference to `mean_c` is replaced by `more_c`.

To demonstrate the relative difficulty of removing the duplicate work for the general case, Listing 3.2 contains the push implementation of the same queries after inlining the definition

```
queries :: IO (Push Record (Int,Int,Int))
queries = do
  more_c ← newIORef 0
  less_c ← newIORef 0
  mean_s ← newIORef 0
  mean_c ← newIORef 0

  let push record = do
    when (open record > close record) $ do
      modifyIORef more_c (+1)
    when (open record < close record) $ do
      modifyIORef less_c (+1)
    when (open record > close record) $ do
      modifyIORef mean_s (+ open record)
      modifyIORef mean_c (+1)

  let done = do
    more_c' ← readIORef more_c
    less_c' ← readIORef less_c
    mean_s' ← readIORef mean_s
    mean_c' ← readIORef mean_c
    return (more_c', less_c', div mean_s' mean_c')

  return (Push push done)
```

Listing 3.2: Push implementation of queries after inlining combinators

of the combinators. In this version, the `more_c` and `mean_c` references both hold the same value, but this fact is only evident with non-local reasoning about the program. The reference initialisations and updates are located in different parts of the program, with potentially interfering writes in-between. We could use a global value numbering (Gulwani and Necula, 2004) algorithm to remove the duplicate work from the push implementation; such algorithms generally require polynomial time in the size of the program. For the benchmarks in Section 3.4 we have twelve queries to fuse together, while some of our production workloads have thousands of queries over the same input. A polynomial time algorithm is unlikely to be practical for such workloads. With the intermediate representation of Icicle we can use a common subexpression elimination algorithm (Chitil, 1997a), which requires $O(n \log n)$ time.

3.3.1 *A more complicated example*

The previous queries were relatively simple to translate to the intermediate language, but the `meanOfLatest` query from Section 3.2 is a bit more involved. That query, again, is:

```
table kvs { key : Date; value : Int }
query meanOfLatest
 = let k    = last  key in
   let avgs = group key of mean value in
   lookup k avgs
```

To convert `meanOfLatest` to a query plan, we must first define the `mean` and `last` functions used in the query. The `mean` function takes a stream of integers and returns the `sum` of the elements divided by the `count`:

```
function mean (e : Element Int)
 = sum e / count;
```

The `last` function uses a fold that initialises the accumulator to the empty date value `NO_DATE`[2], then, for each element, updates it with the date gained from the current element in the stream:

```
function last (d : Element Date)
 = fold l = NO_DATE then d;
```

Inlining these two functions into the `meanOfLatest` query yields the following:

[2] In our production compiler, `last` returns a `Maybe`.

```
query meanOfLatest
 = let lst  = (fold l = NO_DATE then key) in
   let avgs = group key of
              ( (fold s = 0 then s + value)
              / (fold c = 0 then c + 1) ) in
   let ret  = lookup lst avgs
   ret
```

To convert this source query to a plan in the intermediate language, we convert each of the let-bindings separately then concatenate the corresponding parts of the loop anatomy. The `lst` binding becomes a single fold, initialised to `NO_DATE` and updated with the current `key`:

```
plan kvs {      key : Date; value : Int;       }
folds    { fold fL  : Date = NO_DATE then key  }
after    {      lst : Date = fL                }
```

For the avgs binding, each fold accumulator inside the body of the `group` construct is nested within a `group` in the intermediate language. Inside the context of the `group`, the binding for `s` refers to the `Int` value for the current key; outside the `group`, in the `after` section, `s` refers to a value of (`Map Date Int`) containing the values of all keys. Each time we receive a row from the table the accumulator associated with the `key` is updated, using the default value `0` if an entry for that key is not yet present. After we have processed the entire table we join the maps and divide each sum by its corresponding count to yield a map of means for each key.

```
folds  { group key
         { fold s : Int = 0 then s + value
         ; fold c : Int = 0 then c + 1 } }

after  { avgs : Map Date Int
         = mapMap (λsc. fst sc / snd sc) (mapZip s c) }
```

Finally, the `ret` binding from the original query is evaluated in the `after` stage. In the `return` stage we specify that the result of the overall query `avg` is the result of the `ret` binding.

```
  after  { ret : Int = lookup lst avgs }
  return { avg : Int = ret }
```

We then combine the plans from each binding. This query plan is then fused with any other queries that process the same input; the fused query plan is then translated to an imperative loop nest in a similar way to our prior work on flow fusion (Lippmeier et al., 2013). When

translating folds nested inside `filter`s, the update statements are nested inside `if` statements. When translating folds nested inside `group`s, each accumulator becomes a key-value map, and the update statements modify the element corresponding to the current key. As with `filter`s, `group`s with the same key can be merged together. We implement key-value maps as two separate arrays: one array contains the keys, and the other contains the corresponding values. The nested structure of `group`s allows us to re-use the same keys array for multiple accumulators, further reducing duplicate work.

3.4 BENCHMARKS

This section shows the results of benchmarks carried out in 2016. At Ambiata we are using Icicle in production to query medium-sized datasets that fit on a single disk. For larger datasets, we have implemented a scheduler to distribute datasets across multiple nodes and run Icicle on each node separately. The data we are working with is several terabytes compressed which, at the time of benchmarking, would not fit on a single disk. However, each row has a natural primary key and the features we need to compute depend only on the data within single key groups, which makes the workload very easy to distribute.

In our proof of concept testing we replaced an existing R script that performed feature generation with new Icicle code. The R script computed features from a 317GB dataset supplied by a customer, containing records for roughly a million different end-users. For each end-user, the R script computed 12 queries over each of 31 input tables, for 372 query evaluations in total. The R script took 15 hours to run and consisted of 3,566 lines of code. The replacement Icicle version is only 191 lines of code and takes seven minutes to run.

The graph in Figure 3.5 shows the throughput in megabytes per second. We compared the throughput of several programs over the same dataset:

- our original R implementation (R);
- Icicle running single-threaded (1 CPU);
- Icicle running on multiple processors (32 CPU);
- finding empty lines with `grep "^$"` over the same data;
- counting characters, words and lines with `wc`;
- reading and throwing away the results with `cat > /dev/null`.

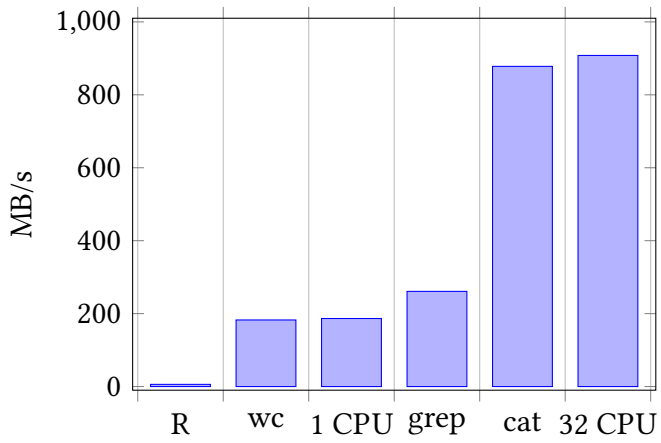


Figure 3.5: Throughput comparisons of Icicle (1 CPU and 32 CPU) against existing R code and standard Unix utilities; higher is faster.

We ran all the Unix utilities with unicode decoding disabled using `LANG=C LC_COLLATE` for maximum performance. The input data does not contain unicode characters. We used an Amazon EC2 `c3.8xlarge` with 32 CPUs, 60GB of RAM, and striped, RAIDed SSD storage. The fused Icicle version significantly outperformed the R version of the queries, and the single-threaded version was on par with `wc`, while only a little slower than `grep`. This is despite the fact that the Icicle queries perform more computational work than `wc` and `grep`. By using multiple processors, we were able to scale up to perform as well as `cat`, approaching the disk speed. The memory usage of Icicle starts at around 200MB of RAM for a single thread, but as more threads are added approaches 15MB per thread. The memory usage is constant in the input size and depends on the number of queries. The R code is single threaded and would require at least 150 processors to reach similar speeds, assuming perfect scaling.

Figure 3.6 shows how the total read throughput scales as the number of fused queries is increased. For each number of queries, we ran two versions of the fused result: one version that wrote the output to disk, and the other that piped the result to `/dev/null`. The graph shows the throughput of the disk version decreasing roughly linearly in the number of queries, while the version ignoring the output remains constant. This suggests that we are IO bound on the write side, as we write the query results for each of the million end-users. The time spent evaluating the queries themselves is small relative to our current IO load.

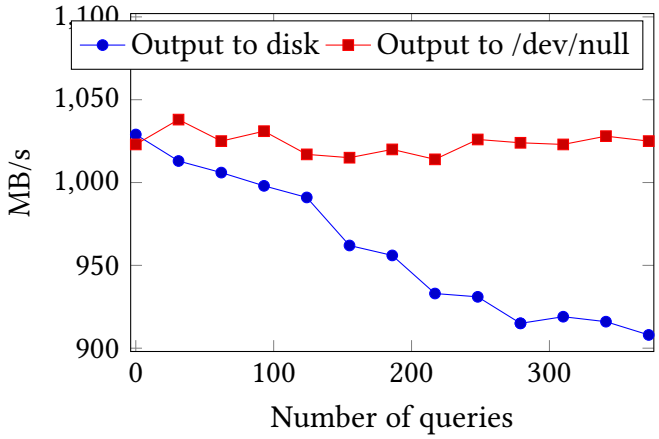


Figure 3.6: Decrease in read throughput as queries are added, comparing writing the output to disk and writing to /dev/null.

3.5 RELATED WORK

In Icicle, as in the push streams from Section 2.3, there is only one input stream, sourced from the input table, which is implicit in the bodies of queries. This approach is intentionally simpler than existing synchronous data flow languages such as Lucy (Mandel et al., 2010) and fusion techniques using synchronous data flow such as flow fusion (Lippmeier et al., 2013). Synchronous data flow languages implement Kahn networks (Vrba et al., 2009) that are restricted to use bounded buffering (Johnston et al., 2004) by clock typing and causal analysis (Stephens, 1997). In such languages, stream combinators with multiple inputs, such as `zip`, are assigned types that require their stream arguments to have the same clock — meaning that elements always arrive in lockstep and the combinators themselves do not need to perform their own buffering. In Icicle the fact that the input stream is implicit and distributed to all combinators means that we can forgo clock analysis. All queries in a program execute in lock-step on the same element at the same moment, which ensures that fusion is a simple matter of concatenating the components of the loop anatomy of each query.

Shortcut fusion techniques such as foldr/build (Gill et al., 1993) and stream fusion (Coutts et al., 2007) rely on inlining to expose fusion opportunities. In Haskell compilers such as GHC, the decision of when to inline is made by internal compiler heuristics, which makes it difficult for the programmer to predict when fusion will occur. When shortcut fusion cannot fuse a program, it fails silently, leaving the programmer unaware of the failure. In this environment, array fusion is considered a “bonus” optimisation rather than integral part of the compilation

method. In contrast, for our feature generation application we really must ensure that multiple queries over the same table are fused, so we cannot rely on heuristics.

StreamIt (Thies et al., 2002) is an imperative streaming language which has been extended with dynamic scheduling (Soule et al., 2013). Dynamic scheduling handles data flow graphs where the transfer rate between different stream operators is not known at compile time. Dynamic scheduling is a trade-off: it is required for stream operators such as grouping and filtering where the output data rate is not known statically, but using dynamic techniques for graphs with static transfer rates tends to have a performance cost. Icicle includes grouping and filtering operators where the output rates are statically unknown, however the associated language constructs require grouped and filtered data to be aggregated rather than passed as the input to another stream operator. This allows Icicle to retain fully static scheduling, so the compiled queries consist of straight line code with no buffering.

Icicle is closely related to work in continuous and shared queries. A continuous query is one that processes input data which may have new records added or removed from it at any time. The result of the continuous query must be updated as soon as the input data changes. Shared queries are ones in which the same sub expressions occur in several individual queries over the same data, and we wish to share the results of these sub expressions among all individuals that use them. For example, in Munagala et al. (2007), input records are filtered by a conjunction of predicates, and the predicates occur in multiple queries. Madden et al. (2002) uses a predicate index to avoid recomputing them. Andrade et al. (2003) describes a compiler for queries over geospacial imagery that shares the results of several pre-defined aggregation functions between queries. Continuous Query Language (CQL) (Arasu et al., 2002; Group et al., 2003) again allows aggregates in its queries, but they must be builtin aggregate functions. Icicle addresses a computationally similar problem, except that our input data sets can only have new records added rather than deleted, which allows us to support general aggregations rather than just filter predicates. It is not obvious how arbitrary aggregate functions could be supported while also allowing deletion of records from the input data — other than by recomputing the entire aggregation after each deletion.

CHAPTER 4

PROCESSES AND NETWORKS

This chapter presents a language for expressing a collection of queries, each with potentially many input streams, as a Kahn process network. This work was first published as Robinson and Lippmeier (2017). The processes in these process networks execute concurrently and communicate via fixed-size bounded buffers between channels. Each buffer is restricted to contain at most a single element. The processes in a process network are then fused together to form a single process which produces the same output streams as the entire network, without the need for inter-process communication.

The contributions of this chapter are:

- We informally introduce processes and fusion with example queries (Section 4.1);
- We present a streaming process calculus with concurrent execution semantics (Section 4.2);
- We present an algorithm for fusing pairs of processes (Section 4.3);
- We motivate an extra synchronisation primitive, `drop`, which coordinates between multiple consumers of the same stream, to improve locality by ensuring both consumers operate on the same value concurrently (Section 4.4);
- We present a heuristic algorithm for fusing an entire process network (Section 4.5);
- We present an overview of the mechanised soundness proofs of fusion (Section 4.6).

4.1 GOLD PANNING WITH PROCESSES

Recall the `priceAnalyses` example from Section 2.1, which performs statistical analyses over the daily prices of a particular corporate stock and market index. Figure 4.1 shows the dependency graph for `priceAnalyses` with the two input streams, `index` and `stock`, at the top of the graph.

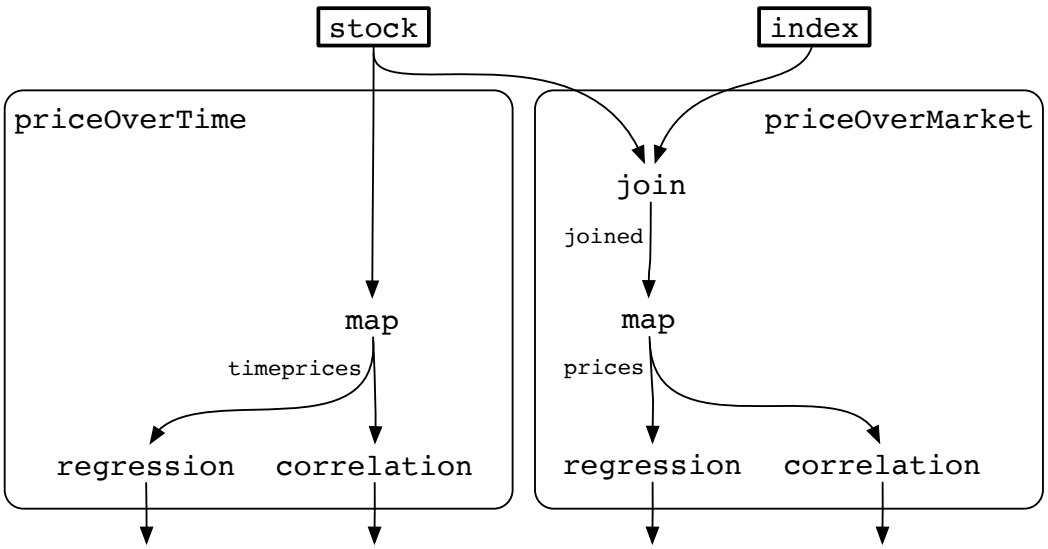


Figure 4.1: Dependency graph for `priceAnalyses` example

As discussed earlier, we cannot execute this example in a single pass using the pull streams from Section 2.2, because the `stock` input stream is used twice, and pull streams only support a single consumer. Similarly, we cannot execute this example in a single pass using the push streams from Section 2.3, because the `join` combinator has two inputs, and push streams only support a single producer except for non-deterministic merge. Rather than just using pull streams, or just using push streams, we wish to be able to perform both pulling and pushing in the same computation, in a way that supports multiple consumers and multiple producers. Kahn process networks (Kahn et al., 1976) are a flexible, expressive way of writing streaming computations, where a network is composed of communicating processes. Executing communicating processes introduces runtime overhead, as stream elements must be passed between processes. Instead, we wish to take this concurrent process network and convert it to sequential code that does not need any runtime scheduling or message passing overhead.

A *process* in our system is a simple imperative program with a local heap. A process pulls source values from an arbitrary number of input streams and pushes result values to at least one output stream. The process language is an intermediate representation we use when fusing the overall dataflow network. When describing the fusion transform we describe the control flow of the process as a state machine.

A *combinator* is a template for a process which parameterises it over the particular input and output streams, as well as values of configuration parameters such as the worker function

```
foldl
  = λ (k  : b → a → b) (z   : b) (j : b → c)
      (sIn: Stream a)   (sOut: Stream c).
    ν (s  : b) (v : a)  (F0..F4: Label).
    process
     { ins:    { sIn  }
     , outs:   { sOut }
     , heap:   { s = z, v }
     , label:    F0
     , instrs: { F0 = pull  sIn     v  F1[] else F2[]
               , F1 = drop  sIn        F0[s = k s v]

               — sIn closed
               , F2 = push  sOut (j s) F3[]
               , F3 = close sOut       F4[]
               , F4 = exit } }
```

Listing 4.1: Process implementation of `foldl`

used in a `map` process. Each process implements a logical *operator* — so we use "operator" when describing the values being computed, but "process" when referring to the implementation.

4.1.1 *Fold combinator*

The definition of the `foldl` combinator, used to implement `correlation` and `regression` in our `priceAnalyses` process network, is given in Listing 4.1. The combinator is parameterised by the fold state update function (`k`) and the fold state initialisation (`z`). In `correlation` and `regression`, the result must be extracted from the fold state; we extend the standard presentation of `foldl` with an eject function (`j`) to perform this extraction. The process reads from an input stream and, at the end of the input stream, produces a single-element output stream containing the fold result. The *nu-binders* (*ν* `(s : a) (v : b)` ...) indicate that each time the `foldl` combinator is instantiated, fresh names must be given to `s`, `v` and so on, that do not conflict with other insantiations. The `s` and `v` bindings refer to variables in the mutable heap of the process. The `s` variable stores the current fold state and is initialised to the initial fold value (`z`); the `v` variable stores the most recent value from the input stream, and is left uninitialised.

The body of the combinator is a record that defines the process. The `ins` field defines the set of input streams, and the `outs` field defines the set of output streams. The `heap` field gives

the initial values of each of the local variables; variables without an explicit initial value are given some arbitrary value. The `instrs` field contains a set of labelled instructions that define the program, while the `label` field gives the label of the initial instruction. In this form, the output stream (`sOut`) is defined via a parameter, rather than being the result of the combinator.

The initial instruction (`pull sIn v F1[] else F2[]`) pulls the next element from the stream `sIn`, writes it into the heap variable `v`, then proceeds to the instruction at label `F1`. The empty list `[]` after the target label `F1` can be used to update heap variables, but as we do not need to update anything yet we leave it empty. If the input stream is finished, there are no more elements to pull; execution proceeds to the instruction at label `F2` instead.

After successfully pulling a new element from the input stream, the instruction at label `F1` (`drop sIn F0[s = k s v]`) signals that the current element that was pulled from stream `sIn` is no longer required, before updating the fold state (`s`) by applying the fold update function (`k`). Execution then proceeds back to the pull instruction at label `F0`. In Section 4.4 we shall see how this `drop` instruction is used to synchronise processes reading from the same shared stream, ensuring that all processes operate on the same element together without overtaking one another.

When the input stream is finished, the instruction (`push sOut (j s) F3[]`) pushes the result of the eject function applied to the final fold state to the output stream `sOut`. Execution then proceeds to the instruction at label `F3`. The comment above the instruction highlights the change in state of the input stream.

Next, the instruction (`close sOut F4[]`) signals that the output stream `sOut` is finished, and then proceeds to the instruction at label `F4`.

Finally, the instruction (`exit`) signals that the process is finished, and has no further work to do. The process terminates.

4.1.2 *Map combinator*

The definition of the `map` combinator, which applies a worker function to every element in the input stream, is given in Listing 4.2. The combinator is parameterised by the worker function (`f`), and takes one input stream (`sIn`) and produces one output stream (`sOut`). The heap variable (`v`) is used to store the last value read from the input stream. The process starts by pulling from the input stream, storing the element in the heap variable (`v`). It then pushes the transformed element (`f v`) into the output stream, drops the element from the input stream,

```
map
  = λ (f  : a → b)
      (sIn: Stream a) (sOut: Stream b).
    ν (v  : a)        (M0..M4: Label).
    process
     { ins:    { sIn  }
     , outs:   { sOut }
     , heap:   { v }
     , label:  M0
     , instrs: { M0 = pull  sIn     v  M1[] else M3[]
               , M1 = push  sOut (f v) M2[]
               , M2 = drop  sIn        M0[]

               -- sIn closed
               , M3 = close sOut       M4[]
               , M4 = exit } }
```

Listing 4.2: Process implementation of `map`

and pulls again. When the input stream finishes, the process closes the output stream and terminates.

4.1.3 *A network of processes*

The `map` and `foldl` combinators are sufficient to express the `priceOverTime` example, which takes a single input stream and computes the correlation and regression. Here is the list implementation of `priceOverTime` again:

```
priceOverTime :: [Record] → (Line, Double)
priceOverTime stock =
  let timeprices = map (λr → (daysSinceEpoch (time r), price r)) stock
  in (regression timeprices, correlation timeprices)
```

We can express `priceOverTime` as a process network by instantiating the above process templates and connecting them together. A process network is a set of processes that are able to communicate with each other.

Listing 4.3 shows the process network for `priceOverTime`. As with the process templates, the network is parameterised by the output streams, which are in this case the output of `regression` and `correlation`. We use the nu-binder syntax to instantiate a fresh name for the `timeprices` internal stream, which is the output of the `map` combinator. We implement

```
priceOverTime =
  λ (stock : Stream Record)
    (reg_out : Stream Line) (cor_out : Stream Double).
  ν (timeprices : Stream (Double,Double)).
     { map    tp_f             stock      timeprices
     , foldl reg_k reg_z reg_j timeprices reg_out
     , foldl cor_k cor_z cor_j timeprices cor_out }
```

Listing 4.3: Process network for `priceOverTime`

`regression` and `correlation` as folds with eject functions. The details of the worker functions given to `map` and `foldl` are defined externally.

In Chapter 3, Icicle used the details of worker functions to perform common subexpression elimination after fusing queries together. We could remove duplicate work from a process after performing fusion if we inlined the definitions of the worker functions into the process. The processes here are more general than Icicle's intermediate language, as is necessary to support both multiple inputs and multiple queries; removing all duplicate work from processes may require a polynomial-time global value numbering algorithm (Gulwani and Necula, 2004) rather than the $O(n \log n)$ common subexpression elimination algorithm. The fusion algorithm itself does not require the details of the worker functions, however, and we leave them externally defined for the present discussion.

4.1.4 *Fusing processes together*

Our fusion algorithm takes two processes and produces a new one that computes the output of both. We fuse a pair of processes in the `priceOverTime` network; to distinguish between the two `foldl` processes in this network, we refer to them as the `regression` and `correlation` processes. As an example, we fuse the `map` process with the `regression` process. The result process computes the result of both processes as if they were executed concurrently, where the output stream of the `map` process is used as the input stream of the `regression` process.

Figure 4.2 shows the result of instantiating the `map` process in the `priceOverTime` process network. The combinator parameters have the corresponding argument value substituted in, and the variables and labels are given fresh names as necessary. We rename the variable name `v` to `tp_v`, to avoid conflict with variables named `v` in other processes. The figure also shows the control flow graph of the process. Figure 4.3 likewise shows the result of instantiating the `regression` process. The instructions and edges in each control flow graph are coloured

```
process — map tp_f stock timeprices
 { ins:    { stock  }
 , outs:   { timeprices }
 , heap:   { tp_v }
 , label:  M0
 , instrs: { M0 = pull  stock       tp_v         M1[] else M3[]
           , M1 = push  timeprices (tp_f tp_v)  M2[]
           , M2 = drop  stock                   M0[]

           — stock closed
           , M3 = close timeprices              M4[]
           , M4 = exit } }
```

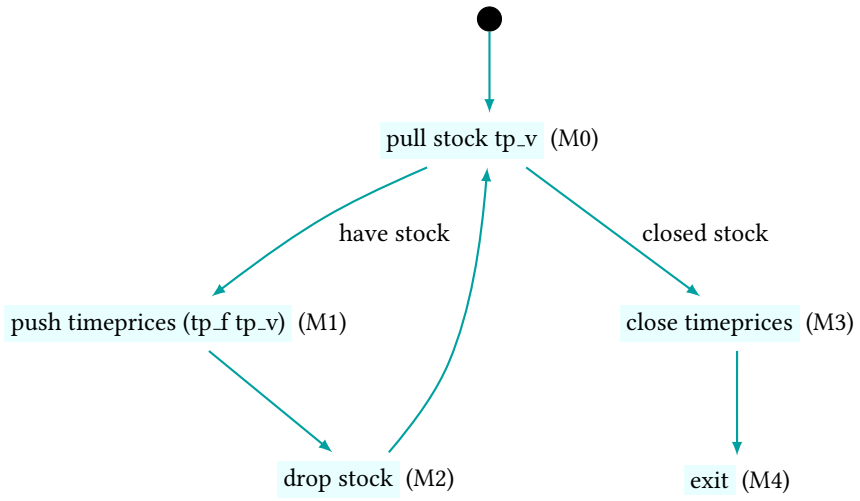


Figure 4.2: Instantiated process for `map` with control flow graph

```
process -- foldl reg_k reg_z reg_j timeprices reg_out
 { ins:    { timeprices  }
 , outs:   { reg_out }
 , heap:   { reg_s = reg_z, reg_v }
 , label:    F0
 , instrs: { F0 = pull  timeprices reg_v  F1[] else F2[]
           , F1 = drop  timeprices        F0[reg_s = reg_k reg_s reg_v]

           -- timeprices closed
           , F2 = push  reg_out (reg_j reg_s) F3[]
           , F3 = close reg_out       F4[]
           , F4 = exit } }
```

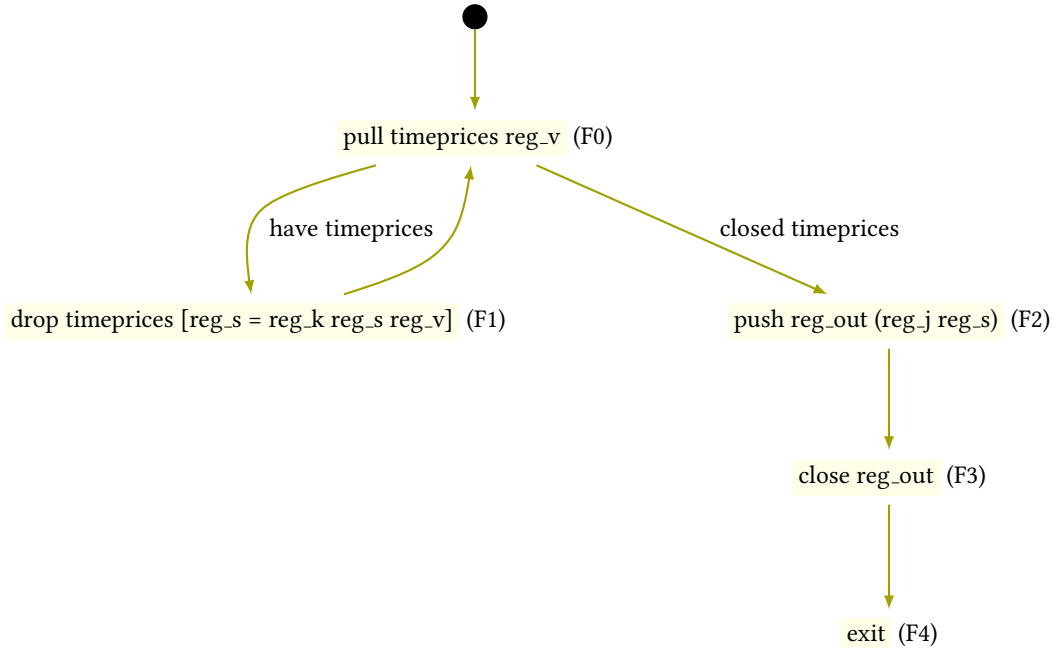


Figure 4.3: Instantiated process for `fold` (`regression`) with control flow graph

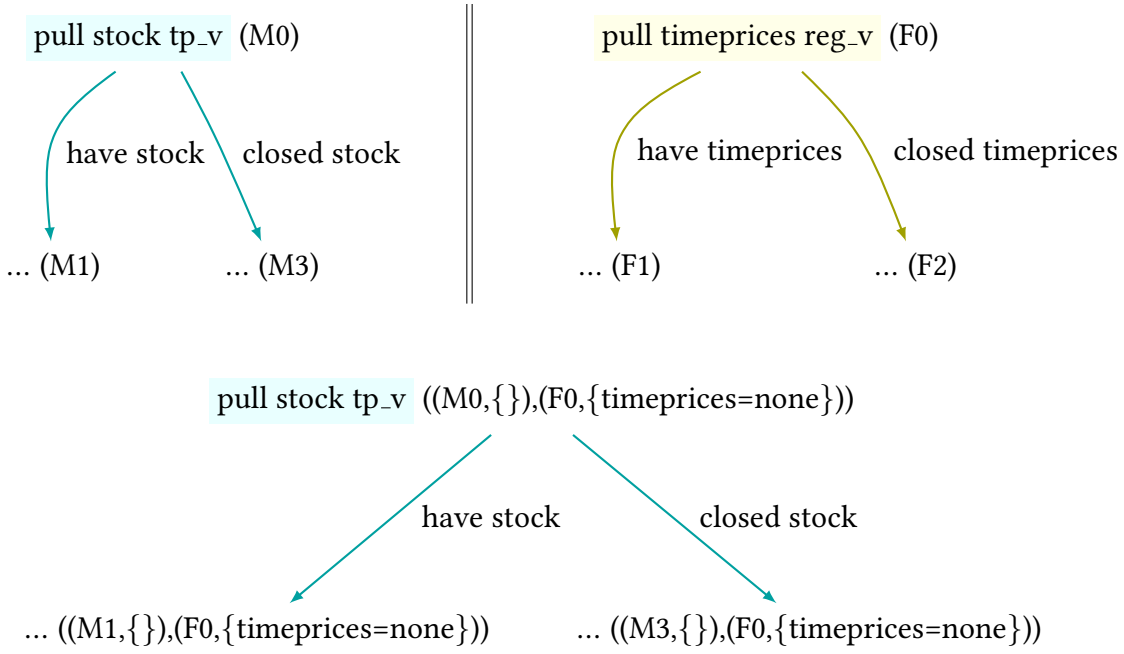


Figure 4.4: Fusing pull instructions for an unshared stream; the left process can pull from the unshared stream, while the right process must wait for the first process to produce a value

differently; the same colours will be used to highlight the provenance of each instruction in our informal description of the fusion algorithm.

Fusing Pulls

The algorithm proceeds by considering pairs of labels and instructions: one from each of the source processes to be fused. First, we consider the initial labels of each process and their corresponding instructions. This situation is shown in Figure 4.4; instructions from the two source processes are shown side-by-side and the instruction of the fused process is below. The `map` process pulls from the `stock` stream, while the `regression` process pulls from the `timeprices` stream. As the `timeprices` stream is produced by the `map` process, the `regression` process must wait until the `map` process pushes a value. If we were to execute the two processes concurrently at this stage, only the `map` process could make progress, by pulling from the `stock` input stream. The corresponding instruction for the fused process pulls from the `stock` input stream, allowing the `map` process to execute while the `regression` process waits.

Each of the joint result labels represents a combination of two source labels, one from each of the source processes. For example, the first joint label ((M0,{}),(F0,{timeprices=none})) represents a combination of the `map` process being in its initial state `M0` and the `regression`

process being in its own initial state `F0`. We also associate each of the joint labels with the *input state*: a description of whether the `regression` process has a value available to read from the shared `timeprices` stream. There is no value available, so the input state for `timeprices` is set to `none`. This extra information only applies to shared input streams; as such, the input state of the `map` process is the empty map.

Fusing Push with Pull

Next, Figure 4.5 shows the `map` process pushing into the `timeprices` stream after pulling a value from the `stock` stream, while the `regression` process is still trying to pull from the `timeprices` stream. After the `map` process pushes a value, this value becomes available for the `regression` process. In the fused process, this situation results in two steps. First, the `map` process pushes the value (`tp_f tp_v`), and stores this value in the new local variable (`chan_tp`), so it is available for the `regression` process. The input state for the `regression` process is updated to (`pending`), to signal that there is a value ready to be pulled in the (`chan_tp`) variable. Next, the `regression` process reads the `pending` value, copying from the (`chan_tp`) variable into the (`reg_v`) variable. The input state for the `regression` process is updated to (`have`), to signal that the `regression` process has copied the pulled value and is using it.

In the original process network, before any fusion, the `timeprices` stream has two consumers: the `regression` and `correlation` processes. Since the fused process implements both `map` and `regression` processes, the fused process still pushes to the `timeprices` stream to allow the `correlation` process to consume it.

Fusion result

Listing 4.4 shows the final result of fusing the `map` and `regression` processes together. There are similar rules for handling the other combinations of instructions, but we defer the details to Section 4.3. The result process has one input stream, `stock`, and two output streams: `timeprices` from `map`, and `reg_out` from `regression`.

To complete the implementation of `priceOverTime`, we would now fuse this result process with the `correlation` process. Note that although the result process has a single shared heap, the heap bindings from each fused process are guaranteed not to interfere, as when we instantiate combinators to create source processes we introduce fresh names.

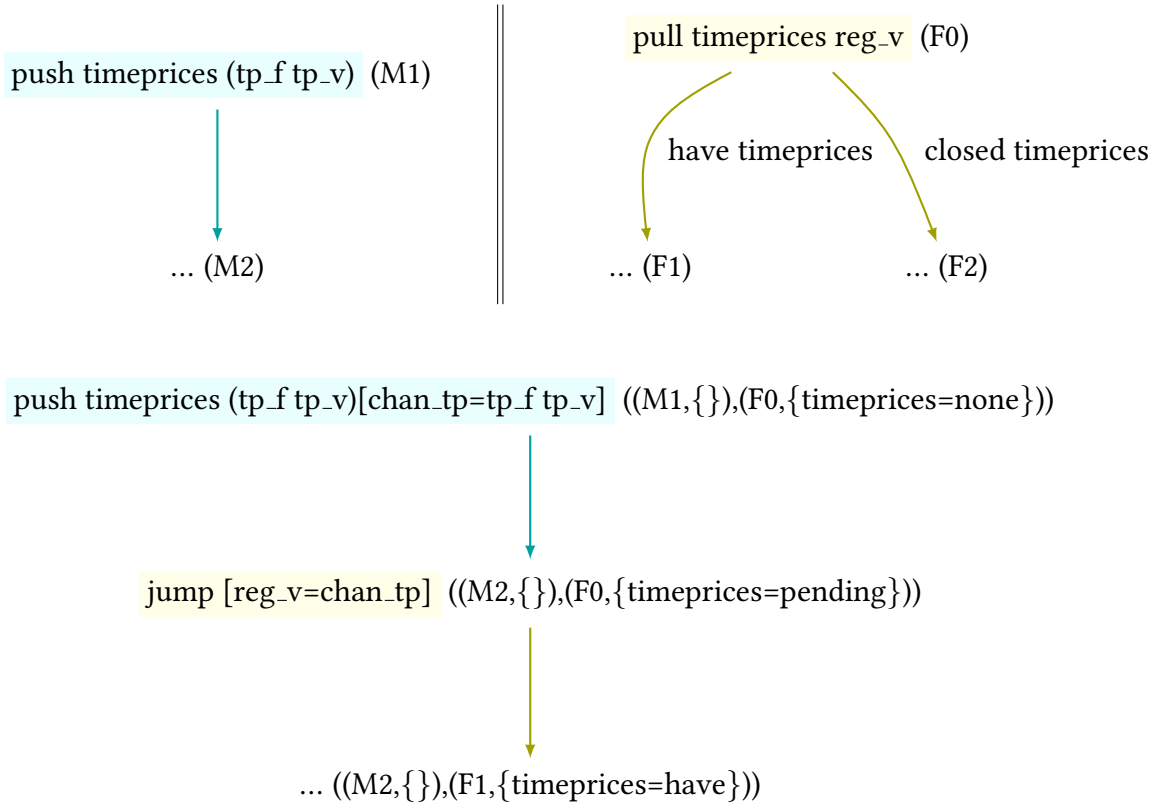


Figure 4.5: Fusing push with pull; the left process produces a value, which the right process consumes

```
process — map tp_f stock timeprices / foldl reg_k reg_z reg_j timeprices reg_out
{ ins:    { stock  }
, outs:   { timeprices
          , reg_out }
, heap:   { tp_v
          , reg_s = reg_z, reg_v
          , chan_tp }
, label:    M0_F0
, instrs:
{ M0_F0   = pull stock tp_v M1_F0[] else M3_F0[]
— ((M0,{}),(F0,{timeprices=none})); (LocalPull)
, M1_F0   = push timeprices (tp_f tp_v) M2_F0_p[chan_tp = (tp_f tp_v)]
— ((M1,{}),(F0,{timeprices=none})); (SharedPush)
, M2_F0_p = jump M2_F1_h[reg_v = chan_tp]
— ((M2,{}),(F0,{timeprices=pending})); (SharedPullPending)
, M2_F1_h = drop stock M0_F1_h[]
— ((M2,{}),(F1,{timeprices=have})); (LocalDrop)
, M0_F1_h = jump M0_F0[reg_s = reg_k reg_s reg_v]
— ((M0,{}),(F1,{timeprices=have})); (ConnectedDrop)

— stock closed
, M3_F0   = close timeprices M4_F0_c[]
— ((M3,{}),(F0,{timeprices=none})); (SharedClose)
, M4_F0_c = jump M4_F2_c[]
— ((M4,{}),(F0,{timeprices=closed})); (SharedPullClosed)
, M4_F2_c = push reg_out (reg_j reg_s) M4_F3_c
— ((M4,{}),(F2,{timeprices=closed})); (LocalPush)
, M4_F3_c = close reg_out M4_F4_c[]
— ((M4,{}),(F3,{timeprices=closed})); (LocalClose)
, M4_F4_c = exit
— ((M4,{}),(F4,{timeprices=closed})); (LocalExit)
} }
```

Listing 4.4: Fusion of `timeprices` and `regression`, along with `shared` instructions and variables

4.1.5 *Join combinator*

To implement the whole `priceAnalyses` process network, we also need the `join` combinator, which pairs together the elements of two sorted input streams. The combinator is parameterised by the key comparison function, which returns an `Ordering` describing whether the key of the first argument is equal to the key of the second argument (`EQ`), lesser (`LT`), or greater (`GT`). The process, shown in Listing 4.5, reads from two input streams (`sA` and `sB`), and produces one output stream (`sOut`). Two heap variables are used to store the most recent input elements (`va` and `vb`), and another is used to store the key comparison (`c`).

In the first group of instructions, the instructions at labels `IN0` and `IN1` pull an element from each input stream. If both pulls are successful, the instruction at label `IN2` compares the input values using the key comparison function (`cmp va vb`). Next, the instruction at label `IN3` checks whether the keys are equal: if so, execution proceeds to the instruction at label `EQ0`; otherwise, instruction proceeds to the instruction at label `NE0`.

The group of instructions at label `EQ0` execute when the element keys are equal, and pushes the pair of elements to the output stream, before dropping both input streams. Execution then proceeds back to the instruction at label `IN0` to pull from the inputs.

The instruction at label `NE0` executes when the element keys are not equal, and proceeds to the instruction at label `LT0` if the first element is lesser, or to the instruction at label `GT0` if the first element is greater. These two groups of instructions drop the input stream with the lesser key and pull a new value from the same input stream, before returning back to the instruction at label `IN2` to compare the elements.

The group of instructions at label `DA0` executes when the `sB` input stream is finished, while the `sA` input stream may still have elements. As with the polarised stream implementation of `join_iii` in Section 2.4, we must drain the leftover elements from the unfinished stream by repeatedly pulling and dropping until there are no more elements. This draining is required because, when fusing processes together, we treat each consumer as having a one-element buffer for each input stream; the producer can only push when all consumers' buffers are empty. Without draining, the producer would be blocked indefinitely on the terminated `join` process, and any other consumers of the stream would be unable to receive input values.

As an alternative to draining, we could extend the process network semantics to include a "disconnect" instruction, which indicates that a process is no longer interested in consuming a particular input stream. Here, we use the simpler process semantics without disconnection, at the expense of having to explicitly drain streams. It may be tempting to instead modify the

```
join
  = λ (cmp : a → b → Ordering)
      (sA  : Stream a) (sB : Stream b)
      (sOut: Stream (a,b)).
    ν (va : a) (vb : b) (c : Ordering) (...: Label).
    process
     { ins:    { sA, sB }
     , outs:   { sOut }
     , heap:   { va, vb, c }
     , label:  IN0
     , instrs: { IN0 = pull  sA va      IN1[] else DB0[]
               , IN1 = pull  sB vb      IN2[] else DA1[]
               , IN2 = jump             IN3[ c = cmp va vb ]
               , IN3 = case  (c == EQ)  EQ0[] else NE0[]

               -- cmp va vb = EQ
               , EQ0 = push  sOut (a,b) EQ1[]
               , EQ1 = drop  sA         EQ2[]
               , EQ2 = drop  sB         IN0[]

               -- cmp va vb ≠ EQ
               , NE0 = case  (c == LT)  LT0[] else GT0[]

               -- cmp va vb = LT
               , LT0 = drop  sA         LT1[]
               , LT1 = pull  sA va      IN2[] else DB1[]

               -- cmp va vb = GT
               , GT0 = drop  sB         GT1[]
               , GT1 = pull  sB vb      IN2[] else DA1[]

               -- sB closed; drain sA
               , DA0 = pull  sA         DA1[] else EX0[]
               , DA1 = drop  sA         DA0[]

               -- sA closed; drain sB
               , DB0 = pull  sB         DB1[] else EX0[]
               , DB1 = drop  sB         DB0[]

               -- sA and sB closed
               , EX0 = close sOut       EX1[]
               , EX1 = exit } }
```

Listing 4.5: Process implementation of `join`

network semantics so that producers do not push to terminated processes, effectively disconnecting the consumer from all inputs upon termination. Such a change to the semantics would allow concurrent execution of unfused process networks with bounded buffers, without the processes having to perform draining. However, a fused result process, which performs the job of two source processes, only terminates once both source processes have terminated; as such, the result process would only disconnect once both source processes have terminated, which may be later than necessary.

The group of instructions at label `DB0` executes when the `sA` input stream is finished, and drains the unfinished `sB` input stream.

Finally, the group of instructions at `EX0` close the output stream and terminate the process.

4.2 PROCESS DEFINITION

The formal grammar for process definitions is given in Figure 4.6. Variables, Channels and Labels are specified by unique names. We refer to the *endpoint* of a stream as a channel. A particular stream may flow into the input channels of several different processes, but can only be produced by a single output channel. For values and expressions we use an untyped lambda calculus with a few primitives. The '||' operator is boolean-or, '+' addition, '/=' not-equal, and '<' less-than. The special `uninitialised` value is used as a default value for uninitialised heap variables, and inhabits every type.

A *Process* is a record with five fields: the `ins` field specifies the input channels; the `outs` field specifies the output channels; the `heap` field specifies the process-local heap; the `label` field specifies the label of the instruction currently being executed; and the `instrs` field specifies a map of labels to instructions. We use the same record when specifying both the definition of a particular process, as well as when giving the evaluation semantics. In the process definition, the `heap` field gives the initial heap of the process, and any variables with unspecified values are assumed to be the `uninitialised` value. The `label` field gives the entry-point in the process definition, though during evaluation it is the label of the instruction currently being executed. Likewise, we usually only list channel names in the `ins` field in the process definition, though during evaluation they are also paired with their current *InputState*. If an *InputState* is not specified we assume it is 'none'.

In the grammar of Figure 4.6, the *InputState* has four options: `none`, which means no value is currently stored in the associated stream buffer variable; (`pending` *Value*), which gives the current value in the stream buffer variable and indicates that it has not yet been copied into a

$$
\begin{array}{lll}
\textit{Exp}, e & ::= & x \mid v \mid e\ e \mid (e \parallel e) \mid e + e \mid e\ \texttt{/=}\ e \mid e < e \\
\textit{Value}, v & ::= & \mathbb{N} \mid \mathbb{B} \mid (\lambda x.\ e) \mid \texttt{uninitialised} \\
\textit{Heap}, bs & ::= & \cdot \mid bs,\ x\ =\ v \\
\textit{Updates}, us & ::= & \cdot \mid us,\ x\ =\ e
\end{array}
$$

$$
\begin{array}{lll}
\textit{Process}, p & ::= & \texttt{process} \\
 & & \quad \texttt{ins:} \quad (\textit{Channel} \mapsto \textit{InputState}) \\
 & & \quad \texttt{outs:} \quad \{\textit{Channel}\} \\
 & & \quad \texttt{heap:} \quad \textit{Heap} \\
 & & \quad \texttt{label:} \quad \textit{Label} \\
 & & \quad \texttt{instrs:}\ (\textit{Label} \mapsto \textit{Instruction})
\end{array}
$$

$$
\textit{InputState} \quad ::= \quad \texttt{none} \mid \texttt{pending}\ \textit{Value} \mid \texttt{have} \mid \texttt{closed}
$$

$$
\begin{array}{lll}
\textit{Variable}, x & \longrightarrow & \text{(value variable)} \\
\textit{Channel}, c & \longrightarrow & \text{(channel name)} \\
\textit{Label}, l & \longrightarrow & \text{(label name)} \\
\textit{ChannelStates} & = & (\textit{Channel} \mapsto \textit{InputState})
\end{array}
$$

$$
\begin{array}{lllllll}
\textit{Instruction} & ::= & \texttt{pull} & \textit{Channel} & \textit{Variable} & \textit{Next} & \textit{Next} \\
 & \mid & \texttt{push} & \textit{Channel} & \textit{Exp} & \textit{Next} & \\
 & \mid & \texttt{close} & \textit{Channel} & & \textit{Next} & \\
 & \mid & \texttt{drop} & \textit{Channel} & & \textit{Next} & \\
 & \mid & \texttt{case} & \textit{Exp} & \textit{Next} & \textit{Next} & \\
 & \mid & \texttt{jump} & & & \textit{Next} & \\
 & \mid & \texttt{exit} & & & &
\end{array}
$$

$$
\textit{Next} \quad = \quad \textit{Label} \times \textit{Updates}
$$

Figure 4.6: Process definitions

process-local variable; `have`, which means the pending value has been copied into a process-local variable; and `closed`, which means the producer has signalled that the channel is finished and will not receive any more values. The *Value* attached to the `pending` state is used when specifying the evaluation semantics of processes. When performing the fusion transform, the *Value* itself will not be known, but we can still reason statically that a process must be in the `pending` state. When defining the fusion transform in Section 4.3, we will use a version of *InputState* with only this statically known information.

The `instrs` field of the *Process* maps labels to instructions. The possible instructions are: `pull`, which tries to pull the next value from a channel into a given heap variable and blocks until the producer pushes a value or closes the channel; `push`, which pushes the value of an expression to an output channel; `close`, which signals the end of an output channel; `drop`, which indicates that the current value pulled from a channel is no longer needed; `case`, which branches based on the result of a boolean expression; `jump`, which causes control to move to a new instruction; and `exit`, which signals that the process is finished.

Instructions include a *Next* field containing the label of the next instruction to execute, as well as a list of $\mathit{Variable} \times \mathit{Exp}$ bindings used to update the heap. The list of update bindings is attached directly to instructions to make the fusion algorithm easier to specify, in contrast to a presentation with a separate `update` instruction.

4.2.1 *Execution*

The dynamic execution semantics for a process network consists of:

1. *Injection* of an action, which can denote an empty message, a pushed channel value, or a channel being closed, into a process or a network. Each individual process only accepts an injected action when it is ready for it, and injection into a network succeeds only when *all* processes accept it.

2. *Advancing* a single process from one state to another, producing an output action. Advancing a network succeeds when *any* of the processes in the network can advance, and the output action can be injected into *all* the other processes.

3. *Feeding* input values from the environment into processes, and collecting outputs of the processes. Feeding alternates between Injecting values from the environment and Advancing the network, until all processes have terminated. When a process pushes

a value to an output channel, we collect this value in a list associated with the output channel.

Execution of a network is non-deterministic. At any moment, several processes may be able to take a step, while others are blocked. As with Kahn processes (Kahn et al., 1976), pulling from a channel is blocking, which enables the overall sequence of values on each output channel to be deterministic. Unlike Kahn processes, pushing to a channel can also block. Each consumer has a single element buffer, and pushing only succeeds when that buffer is empty.

Importantly, it is the order in which values are *pushed to each particular output channel* which is deterministic, whereas the order in which different processes execute their instructions is not. When we fuse two processes, we choose one particular instruction ordering that enables the network to advance without requiring unbounded buffering. The single ordering is chosen by heuristically deciding which pair of states to merge during fusion, and is discussed in Section 4.2.2.

Each channel may be pushed to by a single process only, so in a sense each output channel is owned by a single process. The only intra-process communication is via channels and streams. Our model is "pure data flow" as there are no side-channels between processes — in contrast to "impure data flow" systems such as StreamIt (Thies et al., 2002).

Injection

Figure 4.7 defines the grammar of actions produced by advancing a process in a process network, and gives the rules for injecting these actions into processes. Injection is a meta-level operation, in contrast to `pull` and `push`, which are instructions in the object language. The statement (p; `inject` $a \Rightarrow p'$) reads "given process p, injecting action a yields an updated process p'". An action a is a message describing the state change that can occur to a channel, with three options: ($\cdot$), the empty action, used when a process simply updates internal state; (`push` *Channel Value*), which encodes the value a process pushes to one of its output channels; and (`close` *Channel*), which denotes the end of the stream. The `injects` form is similar to the `inject` form, operating on a process network.

Rule (InjectPush) injects a single value into a single process. The value is stored as a (`pending` v) binding in the *InputState* of the associated channel of the process. The *InputState* acts as a single element buffer, and must be empty (`none`) for injection to succeed. Rule (InjectClose) injects a close message and updates the input state in a similar way.

$$\mathit{Action},\ a \quad ::= \quad \cdot \ \mid\ \mathsf{push}\ \mathit{Channel}\ \mathit{Value} \ \mid\ \mathtt{close}\ \mathit{Channel}$$

$$\boxed{\mathit{Process};\ \mathtt{inject}\ \mathit{Action} \Rightarrow \mathit{Process}}$$

$$\frac{p[\mathtt{ins}][c] = \mathsf{none}}{p;\ \mathtt{inject}\ (\mathsf{push}\ c\ v) \Rightarrow p\ [\mathtt{ins} \mapsto (p[\mathtt{ins}][c \mapsto \mathsf{pending}\ v])]}\ (\text{InjectPush})$$

$$\frac{p[\mathtt{ins}][c] = \mathsf{none}}{p;\ \mathtt{inject}\ (\mathtt{close}\ c) \Rightarrow p\ [\mathtt{ins} \mapsto (p[\mathtt{ins}][c \mapsto \mathtt{closed}])]}\ (\text{InjectClose})$$

$$\frac{c \notin p[\mathtt{ins}]}{p;\ \mathtt{inject}\ (\mathsf{push}\ c\ v) \Rightarrow p}\ (\text{InjectNopPush}) \qquad \frac{c \notin p[\mathtt{ins}]}{p;\ \mathtt{inject}\ (\mathtt{close}\ c) \Rightarrow p}\ (\text{InjectNopClose})$$

$$\frac{}{p;\ \mathtt{inject}\ (\cdot) \Rightarrow p}\ (\text{InjectNopInternal})$$

$$\boxed{\{\mathit{Process}\};\ \mathtt{injects}\ \mathit{Action} \Rightarrow \{\mathit{Process}\}}$$

$$\frac{\{\ p_i;\ \mathtt{inject}\ a \Rightarrow p_i'\ \}^i}{\{p_i\}^i;\ \mathtt{injects}\ a \Rightarrow \{p_i'\}^i}\ (\text{InjectMany})$$

Figure 4.7: Injection of message actions into input channels

Rules (InjectNopPush) and (InjectNopClose) allow processes that do not use a particular named channel to ignore messages injected into that channel. Rule (InjectNopInternal) allows processes to ignore empty messages.

Rule (InjectMany) injects a single value into a network. We use the single process judgment form to inject the value into all processes, which must succeed for all of them. To inject a message into a process network, all the processes which do not ignore the message must be ready to accept the message by having the corresponding *InputState* set to `none`; otherwise, the process would require more than a single-element buffer to store multiple messages.

Advancing

Figure 4.8 gives the rules for advancing a single process and process networks. The statement $(i;\ is;\ bs \overset{a}{\Longrightarrow} l;\ is';\ us')$ reads "instruction i, given channel states is and the heap bindings bs, passes control to instruction at label l and yields new channel states is', heap update expressions us', and performs an output action a."

Rule (PullPending) takes the `pending` value v from the channel state and produces a heap update to copy this value into the variable x in the `pull` instruction. Control is passed to the first output label, l. We use the syntax $(us, x = v)$ to mean that the list of updates us is extended with the new binding $(x = v)$. In the result channel states, the state of the input channel c is updated to `have`, to indicate that the value has been copied into the local variable.

Rule (PullClosed) applies when the channel state is `closed`, passing control to the second output label, l'. As the channel remains closed, there is no need to update the channel state as in the (PullPending) rule.

Rule (Push) evaluates the expression e under heap bindings bs to a value v, and produces a corresponding action which carries this value. The judgment $(bs \vdash e \Downarrow v)$ expresses standard untyped lambda calculus reduction, using the heap bs for the values of free variables. This evaluation is completely standard, and we do not discuss it further.

Rule (Close) emits a `close` action; once injected, this action will transition the recipients' channel states to `closed`. Once a channel is closed it can no longer be pushed to, as the recipients' channel states cannot transition back to the `none` state required by the (InjectPush) rule.

Rule (Drop) changes the input channel state from `have` to `none`. A `drop` instruction can only be executed after `pull` has set the input channel state to `have`.

Rule (Jump) produces a new label and associated update expressions. Rules (CaseT) and (CaseF) evaluate the scrutinee e and emit the appropriate label.

$$\boxed{\mathit{Instruction};\ \mathit{ChannelStates};\ \mathit{Heap} \xRightarrow{\mathit{Action}} \mathit{Label};\ \mathit{ChannelStates};\ \mathit{Updates}}$$

$$\frac{is[c] = \texttt{pending}\ v}{\texttt{pull}\ c\ x\ (l, us)\ (l', us');\ is;\ \Sigma \dot{\Rightarrow} l;\ is[c \mapsto \texttt{have}];\ (us, x = v)}\ (\text{PullPending})$$

$$\frac{is[c] = \texttt{closed}}{\texttt{pull}\ c\ x\ (l, us)\ (l', us');\ is;\ \Sigma \dot{\Rightarrow} l';\ is;\ us'}\ (\text{PullClosed})$$

$$\frac{\Sigma \vdash e \Downarrow v}{\texttt{push}\ c\ e\ (l, us);\ is;\ \Sigma \xRightarrow{\texttt{push}\ c\ v} l;\ is;\ us}\ (\text{Push})$$

$$\frac{}{\texttt{close}\ c\ (l, us);\ is;\ \Sigma \xRightarrow{\texttt{close}\ c} l;\ is;\ us}\ (\text{Close})$$

$$\frac{is[c] = \texttt{have}}{\texttt{drop}\ c\ (l, us);\ is;\ \Sigma \dot{\Rightarrow} l;\ is[c \mapsto \texttt{none}];\ us}\ (\text{Drop}) \qquad \frac{}{\texttt{jump}\ (l, us);\ is;\ \Sigma \dot{\Rightarrow} l;\ is;\ us}\ (\text{Jump})$$

$$\frac{\Sigma \vdash e \Downarrow \texttt{True}}{\texttt{case}\ e\ (l_t, us_t)\ (l_f, us_f);\ is;\ \Sigma \dot{\Rightarrow} l_t;\ is;\ us_t}\ (\text{CaseT})$$

$$\frac{\Sigma \vdash e \Downarrow \texttt{False}}{\texttt{case}\ e\ (l_t, us_t)\ (l_f, us_f);\ is;\ \Sigma \dot{\Rightarrow} l_f;\ is;\ us_f}\ (\text{CaseF})$$

$$\boxed{\mathit{Process} \xRightarrow{\mathit{Action}} \mathit{Process}}$$

$$\frac{p[\texttt{instrs}][p[\texttt{label}]];\ p[\texttt{ins}];\ p[\texttt{heap}] \xRightarrow{a} l;\ is;\ us \qquad p[\texttt{heap}] \vdash us \Downarrow bs}{p \xRightarrow{a} p\ [\texttt{label} \mapsto l,\ \ \texttt{heap} \mapsto (p[\texttt{heap}] \lhd bs),\ \ \texttt{ins} \mapsto is]}\ (\text{Advance})$$

$$\boxed{\{\mathit{Process}\} \xRightarrow{\mathit{Action}} \{\mathit{Process}\}}$$

$$\frac{p_i \xRightarrow{a} p_i' \qquad \forall j \mid j \neq i.\ p_j;\ \texttt{inject}\ a \Rightarrow p_j'}{\{p_0 \ldots p_i \ldots p_n\} \xRightarrow{a} \{p_0' \ldots p_i' \ldots p_n'\}}\ (\text{AdvanceMany})$$

Figure 4.8: Advancing processes

There is no corresponding rule for the `exit` instruction, which denotes a finished process.

The statement ($p \overset{a}{\Rightarrow} p'$) reads "process p advances to new process p', yielding action a". Rule (Advance) advances a single process. We look up the current instruction for the process' `label` and pass it, along with the channel states and heap, to the above single instruction judgment. The update expressions us from the single instruction judgment are reduced to values before updating the heap. We use ($us \lhd bs$) to replace bindings in us with new ones from bs. As the update expressions are pure, the evaluation can be done in any order.

The statement ($ps \overset{a}{\Rightarrow} ps'$) reads "the network ps advances to the network ps', yielding action a". Rule (AdvanceMany) allows an arbitrary, non-deterministically chosen process in the network to advance to a new state while yielding an output action a. For this to succeed, it must be possible to inject the action into all the other processes in the network. As all consuming processes must accept the output action at the time it is created, there is no need to buffer it further in the producing process. When any process in the network produces an output action, we take that as the action of the whole network.

Feeding

Figure 4.9 gives the rules for collecting output actions and feeding external input values to the network. These rules exchange input and output values with the environment in which the network runs.

The statement (i; $ps \Rightarrow o$) reads "when fed input channel values i, network ps executes to termination of all processes, and produces output channel values o". The input channel values map i contains a list of values for each input channel; these channels are inputs of the overall network, and cannot be outputs of any processes. The output channel values map o contains the list of values for every output channel in the network. In a concrete implementation the input and output values would be transported over some IO device, but for the semantics we describe the abstract behavior only.

Rule (FeedExit) terminates execution of a network when all processes have terminated. We require the input channel values map to be empty, instead of allowing the terminated network to ignore any leftover input values. The output channel values map is empty.

Rule (FeedInternal) allows the network to perform local computation in the context of the channel values. This does not affect the input or output values, and execution proceeds with the updated process network.

Rule (FeedPush) collects an output action containing a pushed value (`push` c v) produced by a network. The input is fed to the updated process, which results in output channel map o.

$$\boxed{(Channel \mapsto \overline{Value})\ ;\ \{Process\} \Rightarrow (Channel \mapsto \overline{Value})}$$

$$\frac{\forall p \in ps.\ p[\texttt{instrs}][p[\texttt{label}]] = \texttt{exit}}{\emptyset;\ ps \Rightarrow \emptyset}\ \text{(FeedExit)} \qquad \frac{ps \dot{\Rightarrow} ps' \quad i;\ ps' \Rightarrow o}{i;\ ps \Rightarrow o}\ \text{(FeedInternal)}$$

$$\frac{ps \xRightarrow{\texttt{push}\ c\ v} ps' \quad i;\ ps' \Rightarrow o}{i;\ ps \Rightarrow o[c \mapsto ([v] \mathbin{++} o[c])]}\ \text{(FeedPush)} \qquad \frac{ps \xRightarrow{\texttt{close}\ c} ps' \quad i;\ ps' \Rightarrow o}{i;\ ps \Rightarrow o[c \mapsto []]}\ \text{(FeedClose)}$$

$$\frac{ps;\ \texttt{injects}\ (\texttt{push}\ c\ v) \Rightarrow ps' \quad i[c \mapsto vs];\ ps' \Rightarrow o}{i[c \mapsto ([v] \mathbin{++} vs)];\ ps \Rightarrow o}\ \text{(FeedEnvPush)}$$

$$\frac{ps;\ \texttt{injects}\ (\texttt{close}\ c) \Rightarrow ps' \quad (i \setminus \{c\});\ ps' \Rightarrow o}{i[c \mapsto []];\ ps \Rightarrow o}\ \text{(FeedEnvClose)}$$

Figure 4.9: Feeding process networks

At this point, the output channel map o contains the result of executing the remainder of the process network, after the push has happened. In the output, the pushed value v is added to the start of the list corresponding to the output channel c.

Rule (FeedClose) collects a close output action (`close` c) produced by a network. The output channel map for the channel c is set to the empty list; earlier pushes will prefix elements to this list using rule (FeedPush).

Rule (FeedEnvPush) injects values from the external environment as push messages. The updated process network, after having the value injected, is fed the remainder of the input without the pushed value.

Rule (FeedEnvClose) injects a close message for an external input stream when the corresponding list is empty. When execution continues with the updated process network, the input stream is removed from the channel map using the $(i \setminus \{c\})$ syntax.

4.2.2 *Non-deterministic execution order*

The execution rules of Figure 4.8 and Figure 4.9 are non-deterministic in several ways. Rule (AdvanceMany) allows any process to perform any action at any time, provided all other processes in the network are ready to accept the action; (FeedEnvPush) and (FeedEnvClose) also

allow new values and close messages to be injected from the environment, provided all processes that use the channel are ready to accept the value or close message.

In the semantics, allowing the execution order of processes to be non-deterministic is critical, as it defines a search space where we might find an order that does not require unbounded buffering. For a direct implementation of concurrent processes using message passing and operating system threads, an actual, working, execution order would be discovered dynamically at runtime. In contrast, the role of our fusion system is to construct one of these working orders statically. In the fused result process, the instructions will be scheduled so that they run in one of the orders that would have arisen if the network were executed dynamically. Fusion also eliminates the need to pass messages between processes — once they are fused we can just copy values between heap locations.

4.3 FUSION

Our core fusion algorithm constructs a static execution schedule for a single pair of processes. In Section 4.5.1, we fuse a whole process network by fusing successive pairs of processes until only one remains.

Figure 4.10 defines some auxiliary grammar used during fusion. We extend the *Label* grammar with a new alternative, *LabelF* × *LabelF* for the labels in a fused result process. Each *LabelF* consists of a *Label* from a source process, paired with a map from *Channel* to the statically known part of that channel's current *InputState*. When fusing a whole network, as we fuse pairs of individual processes the labels in the result collect more and more information. Each label of the final, completely fused process encodes the joint state that all the original source processes would be in at that point.

We also extend the existing *Variable* grammar with a (`chan` *c*) form which represents the buffer variable associated with channel `c`. We only need one buffer variable for each channel, and naming them like this saves us from inventing fresh names in the definition of the fusion rules. We used the name (`chan_tp`) back in Section 4.1.4 to avoid introducing a new mechanism at that point in the discussion, when in fact the fused process would use a buffer variable called (`chan timeprices`).

Still in Figure 4.10, *ChannelType2* classifies how channels are used, and possibly shared, between two processes. Type `in2` indicates that both processes `pull` from the same channel, so these actions must be coordinated. Type `in1` indicates that only a single process pulls from the channel. Type `in1out1` indicates that one process pushes to the channel and the other

$$
\begin{array}{lll}
\mathit{Label} & ::= & \ldots \mid \mathit{LabelF} \times \mathit{LabelF} \mid \ldots \\
\mathit{LabelF} & = & \mathit{Label} \times (\mathit{Channel} \mapsto \mathit{InputStateF}) \\
\mathit{InputStateF} & ::= & \mathsf{none}_F \mid \mathsf{pending}_F \mid \mathsf{have}_F \mid \mathsf{closed}_F \\
\mathit{Variable} & ::= & \ldots \mid \mathsf{chan}\ \mathit{Channel} \mid \ldots \\
 & & \\
\mathit{ChannelType}_2 & ::= & \texttt{in2} \mid \texttt{in1} \mid \texttt{in1out1} \mid \texttt{out1}
\end{array}
$$

Figure 4.10: Fusion type definitions.

pulls. Type `out1` indicates that the channel is pushed to by a single process. Each output channel is uniquely owned and cannot be pushed to by more than one process.

Figure 4.11 defines function *fusePair*, which fuses a pair of processes, constructing a result process that does the job of both. We start with a joint label l_0 formed from the initial labels of the two source processes. We then use *tryStepPair* to statically choose which of the two processes to advance, and hence which instruction to execute next. The possible destination labels of that instruction (computed with *outlabels* from Figure 4.14) define new joint labels and reachable states. As we discover reachable states, we add them to a map *bs* of joint label to the corresponding instruction, and repeat the process to a fixpoint where no new states can be discovered.

Figure 4.12 defines function *tryStepPair*, which decides which of the two input processes to advance. It starts by calling *tryStep* for both processes. If both can advance, we use heuristics to decide which one to run first.

Clauses (DeferExit1) and (DeferExit2) ensure that the fused process only terminates once both processes are ready to terminate; if either has remaining work, the process with remaining work will execute. The clauses achieve this by checking if either process is at an `exit` instruction, and if so, choosing the other process. The instruction for the second process was computed by calling *tryStep* with the label arguments swapped, so in (DeferExit2) we need to swap the labels back with *swaplabels* (from Figure 4.14). The result process terminates once both processes have terminated at an `exit` instruction; in this case, clause (DeferExit1) will return the `exit` instruction from the first process.

Clauses (PreferJump1) and (PreferJump2) prioritise processes that can perform a `jump`. This helps collect `jump` instructions together so they are easier for post-fusion optimisation to handle (Section 4.5).

```
fusePair : Process → Process → Maybe Process
fusePair p q
  | Just is ← go {} l0
  = Just (process
              ins:  {c | c = t ∈ cs, t ∈ {in1, in2}}
             outs:  {c | c = t ∈ cs, t ∈ {in1out1, out1}}
             heap:  p[heap] ∪ q[heap]
            label:  l0
           instrs:  is)
  | otherwise = Nothing
 where
  cs   = channels p q
  l0   = ( (p[label], {c = none_F | c ∈ p[ins]})
         , (q[label], {c = none_F | c ∈ q[ins]}))
  go bs (lp, lq)
       | (lp, lq) ∈ bs
       = Just bs
       | Just b ← tryStepPair cs lp p[instrs][lp] lq q[instrs][lq]
       = foldM go (bs ∪ {(lp, lq) = b}) (outlabels b)
       | otherwise = Nothing
```

Figure 4.11: Fusion of pairs of processes

```
tryStepPair : (Channel ↦ ChannelType2)
            → LabelF → Instruction → LabelF → Instruction
            → Maybe Instruction
tryStepPair cs l_p i_p l_q i_q =
  match (tryStep cs l_p i_p l_q, tryStep cs l_q i_q l_p) with
  (Just i'_p, Just i'_q)
    | exit ← i'_q        → Just i'_p                (DeferExit1)
    | exit ← i'_p        → Just (swaplabels i'_q)   (DeferExit2)
    | jump _ ← i'_p      → Just i'_p                (PreferJump1)
    | jump _ ← i'_q      → Just (swaplabels i'_q)   (PreferJump2)
    | pull _ _ _ _ ← i'_q → Just i'_p                (DeferPull1)
    | pull _ _ _ _ ← i'_p → Just (swaplabels i'_q)   (DeferPull2)
  (Just i'_p, _)         → Just i'_p                (Run1)
  (_, Just i'_q)         → Just (swaplabels i'_q)   (Run2)
  (Nothing, Nothing)     → Nothing                  (Deadlock)
```

Figure 4.12: Fusion step coordination for a pair of processes.

Similarly, clauses (DeferPull1) and (DeferPull2) defer `pull` instructions: if one of the instructions is a `pull`, we advance the other one. We do this because `pull` instructions may block, while other instructions are more likely to produce immediate results.

Clauses (Run1) and (Run2) apply when the above heuristics do not apply, or only one of the processes can advance.

Clause (Deadlock) applies when neither process can advance, in which case the processes cannot be fused together and fusion fails.

Figure 4.13 defines function *tryStep*, which schedules a single instruction. This function takes the map of channel types, along with the current label and associated instruction of the first (left) process, and the current label of the other (right) process. The *tryStep* function is called twice in *tryStepPair*, once for each process, so the left process may correspond to either input process at any given time.

Clause (LocalJump) applies when the left process wants to jump. In this case, the result instruction simply performs the corresponding jump, leaving the right process where it is. This clause corresponds to a static version of the rule (Jump) for advancing processes during execution (Section 4.2.1).

Clause (LocalCase) is similar, except there are two *Next* labels.

$\mathit{tryStep} \;:\; (\mathit{Channel} \mapsto \mathit{ChannelType2}) \to \mathit{LabelF} \to \mathit{Instruction} \to \mathit{LabelF} \to \mathit{Maybe\ Instruction}$
$\mathit{tryStep}\ \mathit{cs}\ (l_p, s_p)\ i_p\ (l_q, s_q) \;=\; \mathtt{match}\ i_p\ \mathtt{with}$

$\mathtt{jump}\ (l', u')$ (LocalJump)
$\quad \to\ \mathtt{Just}\ (\mathtt{jump}\ ((l',\ s_p),\ (l_q,\ s_q),\ u'))$

$\mathtt{case}\ e\ (l'_t, u'_t)\ (l'_f, u'_f)$ (LocalCase)
$\quad \to\ \mathtt{Just}\ (\mathtt{case}\ e\ ((l'_t,\ s_p),\ (l_q,\ s_q),\ u'_t)\ ((l'_f,\ s_p),\ (l_q,\ s_q),\ u'_f))$

$\mathtt{push}\ c\ e\ (l', u')$
$\quad \mid \mathit{cs}[c] = \mathtt{out1}$ (LocalPush)
$\quad \to\ \mathtt{Just}\ (\mathtt{push}\ c\ e\ ((l',\ s_p),\ (l_q,\ s_q),\ u'))$
$\quad \mid \mathit{cs}[c] = \mathtt{in1out1}\ \wedge\ s_q[c] = \mathtt{none}_F$ (SharedPush)
$\quad \to\ \mathtt{Just}\ (\mathtt{push}\ c\ e\ ((l',\ s_p),\ (l_q,\ s_q[c\ \mapsto\ \mathtt{pending}_F]),\ u'[\mathtt{chan}\ c\ \mapsto\ e]))$

$\mathtt{pull}\ c\ x\ (l'_o, u'_o)\ (l'_c, u'_c)$
$\quad \mid \mathit{cs}[c] = \mathtt{in1}$ (LocalPull)
$\quad \to\ \mathtt{Just}\ (\mathtt{pull}\ c\ x\ ((l'_o,\ s_p),\ (l_q,\ s_q),\ u'_o)\ ((l'_c,\ s_p),\ (l_q,\ s_q),\ u'_c))$
$\quad \mid (\mathit{cs}[c] = \mathtt{in2} \vee \mathit{cs}[c] = \mathtt{in1out1})\ \wedge\ s_p[c] = \mathtt{pending}_F$ (SharedPullPending)
$\quad \to\ \mathtt{Just}\ (\mathtt{jump}\ ((l'_o,\ s_p[c\ \mapsto\ \mathtt{have}_F]),\ (l_q,\ s_q),\ u'_o[x\ \mapsto\ \mathtt{chan}\ c]))$
$\quad \mid (\mathit{cs}[c] = \mathtt{in2} \vee \mathit{cs}[c] = \mathtt{in1out1})\ \wedge\ s_p[c] = \mathtt{closed}_F$ (SharedPullClosed)
$\quad \to\ \mathtt{Just}\ (\mathtt{jump}\ ((l'_c,\ s_p),\ (l_q,\ s_q),\ u'_c))$
$\quad \mid \mathit{cs}[c] = \mathtt{in2}\ \wedge\ s_p[c] = \mathtt{none}_F\ \wedge\ s_q[c] = \mathtt{none}_F$ (SharedPullInject)
$\quad \to\ \mathtt{Just}\ (\mathtt{pull}\ c\ (\mathtt{chan}\ c)$
$\qquad\qquad ((l_p,\ s_p[c\ \mapsto\ \mathtt{pending}_F]),\ (l_q,\ s_q[c\ \mapsto\ \mathtt{pending}_F]),\ [])$
$\qquad\qquad ((l_p,\ s_p[c\ \mapsto\ \mathtt{closed}_F]),\ (l_q,\ s_q[c\ \mapsto\ \mathtt{closed}_F]),\ []))$

$\mathtt{drop}\ c\ (l', u')$
$\quad \mid \mathit{cs}[c] = \mathtt{in1}$ (LocalDrop)
$\quad \to\ \mathtt{Just}\ (\mathtt{drop}\ c\ ((l',\ s_p),\ (l_q,\ s_q),\ u'))$
$\quad \mid \mathit{cs}[c] = \mathtt{in1out1}$ (ConnectedDrop)
$\quad \to\ \mathtt{Just}\ (\mathtt{jump}\ ((l',\ s_p[c\ \mapsto\ \mathtt{none}_F]),\ (l_q,\ s_q),\ u'))$
$\quad \mid \mathit{cs}[c] = \mathtt{in2}\ \wedge\ (s_q[c] = \mathtt{have}_F \vee s_q[c] = \mathtt{pending}_F)$ (SharedDropOne)
$\quad \to\ \mathtt{Just}\ (\mathtt{jump}\ ((l',\ s_p[c\ \mapsto\ \mathtt{none}_F]),\ (l_q,\ s_q),\ u'))$
$\quad \mid \mathit{cs}[c] = \mathtt{in2}\ \wedge\ s_q[c] = \mathtt{none}_F$ (SharedDropBoth)
$\quad \to\ \mathtt{Just}\ (\mathtt{drop}\ c\ ((l',\ s_p[c\ \mapsto\ \mathtt{none}_F]),\ (l_q,\ s_q),\ u'))$

$\mathtt{close}\ c\ (l', u')$
$\quad \mid \mathit{cs}[c] = \mathtt{out1}$ (LocalClose)
$\quad \to\ \mathtt{Just}\ (\mathtt{close}\ c\ ((l',\ s_p),\ (l_q,\ s_q),\ u'))$
$\quad \mid \mathit{cs}[c] = \mathtt{in1out1}\ \wedge\ s_q[c] = \mathtt{none}_F$ (SharedClose)
$\quad \to\ \mathtt{Just}\ (\mathtt{close}\ c\ ((l',\ s_p),\ (l_q,\ s_q[c\ \mapsto\ \mathtt{closed}_F]),\ u'))$

$\mathtt{exit}$ (LocalExit)
$\quad \to\ \mathtt{Just\ exit}$

$\_ \mid \mathtt{otherwise}$ (Blocked)
$\quad \to\ \mathtt{Nothing}$

Figure 4.13: Fusion step for a single process of the pair.

Clause (LocalPush) applies when the left process wants to push to a non-shared output channel. In this case the push can be performed directly, with no additional coordination required.

Clause (SharedPush) applies when the left process wants to push to a shared channel. Pushing to a shared channel requires the downstream process to be ready to accept the value at the same time. We encode this constraint by requiring the static input state of the downstream channel to be $\texttt{none}_F$. When this constraint is satisfied, the result instruction stores the pushed value in the stream buffer variable ($\texttt{chan}\ c$) and sets the static input state to $\texttt{pending}_F$, which indicates that the new value is now available. This clause corresponds to a static version of the evaluation rule (Push) for advancing the left process, combined with the rule (InjectPush) for injecting the push action into the right process.

Still in Figure 4.13, clause (LocalPull) applies when the left process wants to pull from a local channel, which requires no coordination.

Clause (SharedPullPending) applies when the left process wants to pull from a shared channel that the other process either pulls from or pushes to. We know that there is already a value in the stream buffer variable, because the state for that channel is $\texttt{pending}_F$. The result instruction copies the value from the stream buffer variable into a variable specific to the left source process. The corresponding $\texttt{have}_F$ channel state in the result label records that the value has been successfully pulled.

Clause (SharedPullClosed) applies when the left process wants to pull from a shared channel that the other process either pulls from or pushes to, and the channel is closed. The result instruction jumps to the close output label.

Clause (SharedPullInject) applies when the left process wants to pull from a shared channel that both processes pull from, and neither already has a value. The result instruction is a `pull` that loads the stream buffer variable, leaving the labels the same and updating the channel state for both processes. In the next instruction, the left process will try to pull again with the updated channel state, and one of the clauses (SharedPullPending) or (SharedPullClosed) will apply.

Clause (LocalDrop) applies when the left process wants to drop the current value that it read from an unshared input channel, which requires no coordination.

Clause (ConnectedDrop) applies when the left process wants to drop the current value that it received from an upstream process. As the value will have been sent via a heap variable instead of a still extant channel, the result instruction just performs a `jump` while updating the static channel state.

Clauses (SharedDropOne) and (SharedDropBoth) apply when the left process wants to drop from a channel shared by both processes. In (SharedDropOne), the channel states reveal that the other process is still using the value. In this case, the result is a `jump` updating the channel state to note that the left process has dropped. In (SharedDropBoth), the channel states reveal that the other process has already dropped its copy of the channel value using clause (SharedDropOne). In this case, the result is a real `drop`, because we are sure that neither process requires the value any longer.

Clause (LocalClose) applies when the left process wants to close an unshared output channel, which requires no coordination.

Clause (SharedClose) applies when the left process wants to close a shared output channel that the other process pulls from. Closing the channel updates the channel state and requires the downstream process to have dropped any previously read values, just as the (InjectClose) evaluation rule requires the downstream process to have `none` as its input state.

Clause (LocalExit) applies when the left process wants to finish execution, which requires no coordination here, but causes the other process to be prioritised in the (DeferExit) clauses in the earlier definition of *tryStepPair*.

Clause (Blocked) returns `Nothing` when no other clauses apply, meaning that this process is waiting for the other process to advance.

All the clauses in the *tryStep* function work together to perform a static version of the dynamic process execution. Each clause checks whether the left process can advance given the statically known channel state. When the left process advances normally in the dynamic execution rules, it produces an output action to be injected into other processes in the network. The clauses in *tryStep* statically coordinate with the right process, checking whether the result action from advancing the left process can be injected into the right process, given the statically known channel state of the right process. Although there are many clauses, the translation from the advance and injection rules to the clauses is relatively straightforward.

Figure 4.14 contains definitions of some utility functions which we have already mentioned. Function *channels* computes the $ChannelType_2$ map for a pair of processes. Function *outlabels* gets the set of output labels for an instruction, which is used when computing the fixpoint of reachable states. Function *swaplabels* flips the order of the compound labels in an instruction.

$$
\begin{array}{lcl}
\mathit{channels} & : & \mathit{Process} \to \mathit{Process} \to (\mathit{Channel} \mapsto \mathit{ChannelType}_2) \\
\mathit{channels}\ p\ q & = & \{c = \mathtt{in2} \mid c \in (\mathtt{ins}\ p \cap \mathtt{ins}\ q)\} \\
& \cup & \{c = \mathtt{in1} \mid c \in (\mathtt{ins}\ p \cup \mathtt{ins}\ q) \wedge c \notin (\mathtt{outs}\ p \cup \mathtt{outs}\ q)\} \\
& \cup & \{c = \mathtt{in1out1} \mid c \in (\mathtt{ins}\ p \cup \mathtt{ins}\ q) \wedge c \in (\mathtt{outs}\ p \cup \mathtt{outs}\ q)\} \\
& \cup & \{c = \mathtt{out1} \mid c \notin (\mathtt{ins}\ p \cup \mathtt{ins}\ q) \wedge c \in (\mathtt{outs}\ p \cup \mathtt{outs}\ q)\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{outlabels} & : & \mathit{Instruction} \to \{\mathit{Label}\} \\
\mathit{outlabels}\ (\mathtt{pull}\ c\ x\ (l,u)\ (l',u')) & = & \{l, l'\} \\
\mathit{outlabels}\ (\mathtt{drop}\ c\ (l,u)) & = & \{l\} \\
\mathit{outlabels}\ (\mathtt{push}\ c\ e\ (l,u)) & = & \{l\} \\
\mathit{outlabels}\ (\mathtt{close}\ c\ (l,u)) & = & \{l\} \\
\mathit{outlabels}\ (\mathtt{case}\ e\ (l,u)\ (l',u')) & = & \{l, l'\} \\
\mathit{outlabels}\ (\mathtt{jump}\ (l,u)) & = & \{l\} \\
\mathit{outlabels}\ (\mathtt{exit}) & = & \{\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{swaplabels} & : & \mathit{Instruction} \to \mathit{Instruction} \\
\mathit{swaplabels}\ (\mathtt{pull}\ c\ x\ ((l_1,l_2),u)\ ((l'_1,l'_2),u')) & = & \mathtt{pull}\ c\ x\ ((l_2,l_1),u)\ ((l'_2,l'_1),u') \\
\mathit{swaplabels}\ (\mathtt{drop}\ c\ ((l_1,l_2),u)) & = & \mathtt{drop}\ c\ ((l_2,l_1),u) \\
\mathit{swaplabels}\ (\mathtt{push}\ c\ e\ ((l_1,l_2),u)) & = & \mathtt{push}\ c\ e\ ((l_2,l_1),u) \\
\mathit{swaplabels}\ (\mathtt{close}\ c\ ((l_1,l_2),u)) & = & \mathtt{close}\ c\ ((l_2,l_1),u) \\
\mathit{swaplabels}\ (\mathtt{case}\ e\ ((l_1,l_2),u)\ ((l'_1,l'_2),u')) & = & \mathtt{case}\ e\ ((l_2,l_1),u)\ ((l'_2,l'_1),u') \\
\mathit{swaplabels}\ (\mathtt{jump}\ ((l_1,l_2),u)) & = & \mathtt{jump}\ ((l_2,l_1),u) \\
\mathit{swaplabels}\ (\mathtt{exit}) & = & \mathtt{exit}
\end{array}
$$

Figure 4.14: Utility functions for fusion

4.3.1 *Static deadlock detection*

In the definition of *tryStep*, clause (Blocked) applies when the source process is waiting for the other process to advance. If both processes are blocked waiting for each other, then the two processes are stuck. Clause (Deadlock) from *tryStepPair* applies, and fusion fails.

This fusion failure corresponds to a static approximation of deadlock detection. As observed by Buck and Lee (1993), static deadlock detection for Kahn process networks is in general undecidable. Our deadlock detection is sound, but not complete. If we detect a deadlock statically, then concurrent execution of the original Kahn process network may or may not deadlock at runtime. If we do not detect a deadlock, then the original Kahn process network is guaranteed to be free from deadlocks.

Deadlock detection algorithms for Kahn process networks do exist, but perform deadlock detection dynamically rather than statically (Allen et al., 2007; Jiang et al., 2008). These algorithms tend to focus on *artificial deadlocks*, which are deadlocks introduced by restricting the size of channel buffers. Artificial deadlocks are identified dynamically, and resolved by increasing the buffer size (Geilen and Basten, 2003; Parks, 1995). Because these systems perform dynamic scheduling and deadlock detection, process networks using these systems tend to require larger, more coarse-grained processes to offset the dynamic scheduling overhead (Chen et al., 1990).

4.4 SYNCHRONISING PULLING BY DROPPING

The `drop` instruction exists to synchronise two consumers of the same input, so that both consumers pull the same value at roughly the same time. When fusing two consumers, the fusion algorithm uses drops when coordinating the consumers to ensure that one consumer cannot start processing the next element until the other has finished processing the current element. Drop instructions are not necessary for correctness, or for ensuring boundedness of buffers, but they improve locality in the fused process.

Recall the `priceOverTime` example, which computes the correlation and regression of an input stream. The dependency graph for `priceOverTime` is shown in Figure 4.15.

Figure 4.16 shows an example execution of the `correlation` and `regression` processes from `priceOverTime`, with an input stream containing two elements. Execution is displayed as a sequence diagram. Each process and output stream is represented as a vertical line which communicates with other processes by messages, represented by arrows. The names of each

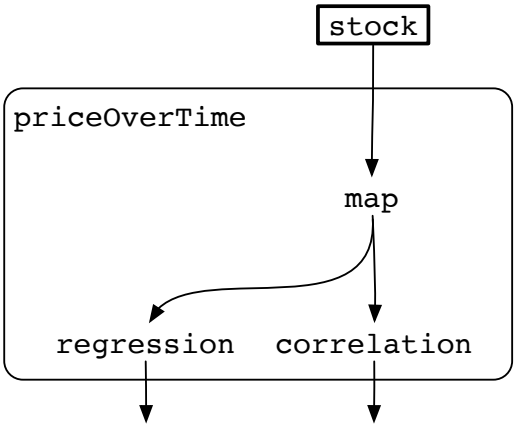


Figure 4.15: Dependency graph for `priceOverTime` example

stream and process are written above the line, and time flows downwards. To highlight the synchronisation between `regression` and `correlation` processes, we use placeholder values such as *A* and *B* instead of actual stream values, and show only a subset of the whole execution, omitting the `stock` input stream and the internal messages of the `map` process.

In the definition of the `fold` process template, the update binding attached to the `drop` instruction updates the fold state with the most recently pulled value. We use the shorthand (drop [update *A*]) to signify that the process updates its fold state with the pulled value *A* after dropping the element.

Execution starts with the `map` process pushing the value *A* to both of its consumers, the `regression` and `correlation` processes. In the execution semantics from Section 4.2.1, this push changes the input state of each recipient process from `none` to `pending`, to signify that there is a value available to pull. At this point, the `map` process cannot push again until both consumers have pulled and dropped the *A* value. Next, the `regression` process pulls the value *A*, temporarily changing its input state to `have`, before using and dropping the value, changing the input state back to `none`. In the execution semantics, the pull instruction updates the process' local state, but does not communicate with any other process; the diagram shows the pull instruction as the process sending a message to itself. The `correlation` process now performs the same. After both consumers have dropped the input value, the `map` process is able to push the next value, *B*, which the consumers operate on similarly. Finally, the `map` process sends close messages to both consumers, which both push the results of the folds to their corresponding output streams before closing them. In this execution, both consumer processes transform the same element at roughly the same time, because the next element is only available once both have dropped, thereby agreeing to accept the next element.

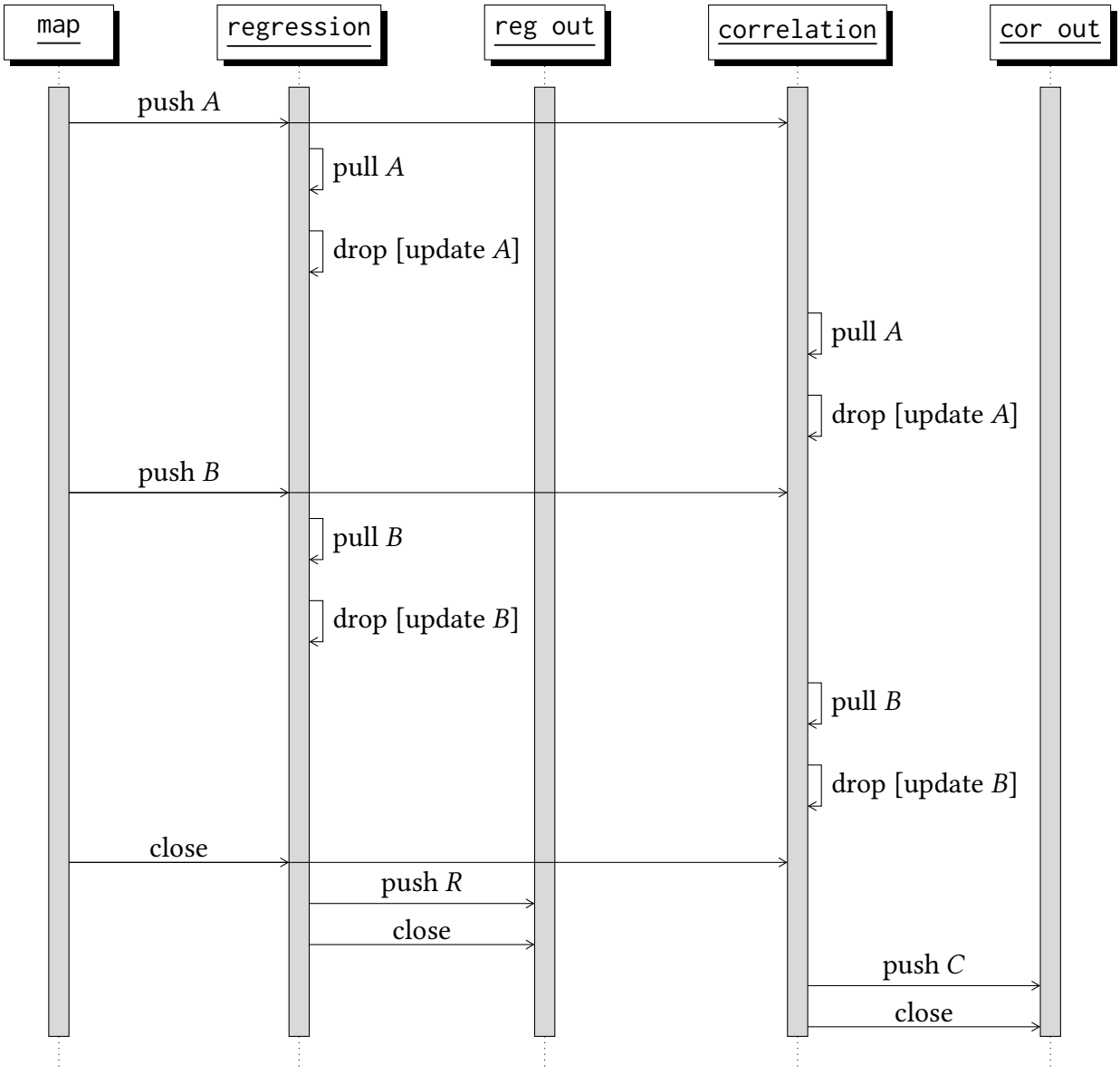


Figure 4.16: Sequence diagram of a possible linearised execution of `priceOverTime`, showing drop synchronisation

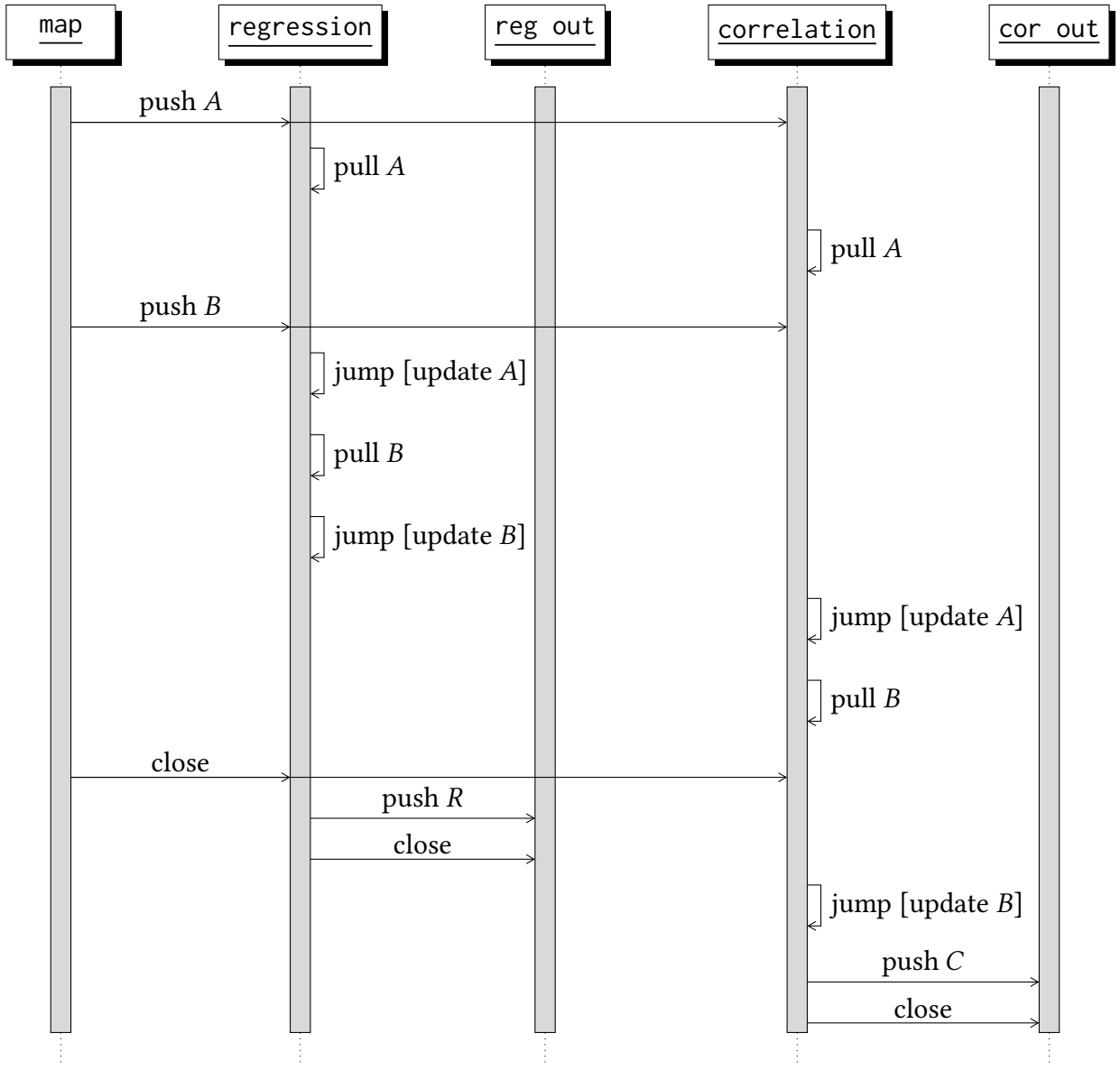


Figure 4.17: Sequence diagram for a possible linearised execution of `priceOverTime`, using a hypothetical semantics without drop synchronisation

Figure 4.17 shows a hypothetical execution which may occur if our process network semantics did not use drop instructions to synchronise between all consumers. In this execution, we replace the drop instructions with a jump instruction, using the shorthand (jump [update A]) to signify updating the fold state. As before, execution starts with the `map` process pushing the A value, which both consumers pull. Without drop synchronisation, the single-element buffer is cleared as soon as the process pulls, allowing the `map` process to push another element to both consumers. Now, the `regression` process executes and updates the fold state with both values A and B. The `regression` process has consumed both elements before the second consumer, `correlation`, has even looked at the first. This is not a problem for concurrent execution: the execution results in the same value, and the buffer is still bounded, containing at most one element. However, when we fuse this network into a single process, we commit to a particular interleaving of execution of the processes in the network. When performing fusion, we would prefer to use the previous interleaving with drop synchronisation to this unsynchronised interleaving, because the process with the unsynchronised interleaving would need to keep track of two consecutive elements at the same time. Keeping both elements means the process requires more live variables, which makes it less likely that both elements will fit in the available registers or cache when we eventually convert the fused process to machine code.

4.5 TRANSFORMING PROCESS NETWORKS

The fusion algorithm described in Section 4.3 operates on a pair of source processes. For process networks that contain more than two processes, we repeatedly fuse pairs of processes in the network together until only one process remains, or until no more fusion is possible. The remaining process performs the job of the entire original process network. When the process network contains a producer with multiple consumers, the result process coordinates between all the consumers to reach consensus on when the producer can push the next value. We use global knowledge of the network to coordinate with all consumers statically, confident that we have not missed any consumers. In contrast, shortcut fusion systems such as (Gill et al., 1993) use rewrite rules to remove intermediate buffers and require only local knowledge, looking at each edge in the network in isolation, but cannot coordinate between multiple consumers. In cases where shortcut fusion cannot fuse it fails silently, leaving the programmer unaware of the failure. This silence is also due to the local nature of rewrite rules: to know whether all the processes have been fused, we need to know about all the processes.

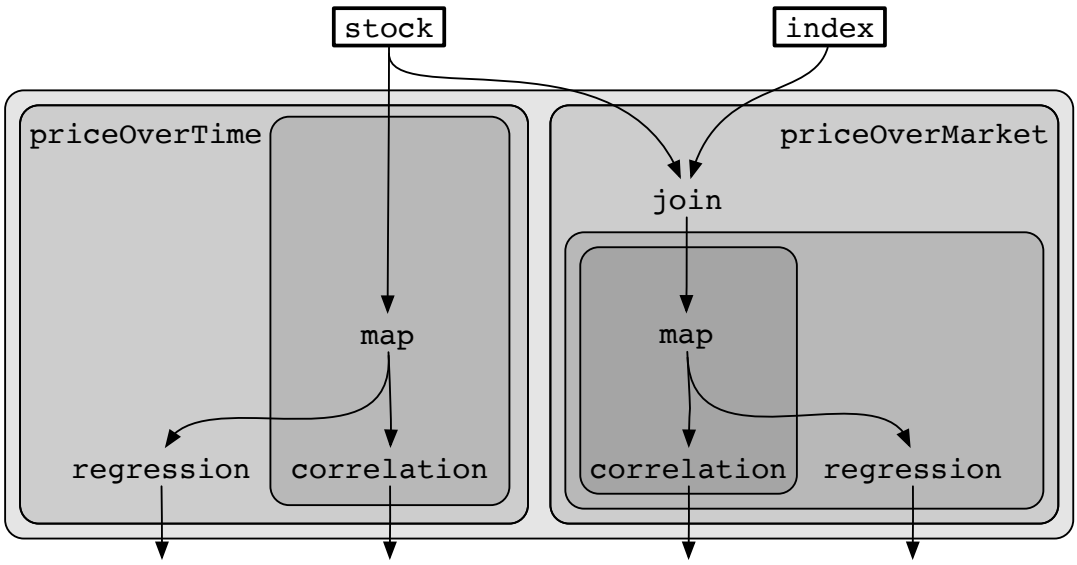


Figure 4.18: Pairwise fusion ordering of the `priceAnalyses` network

When fusing a pair of processes, the fused result process tends to have more states than each source process individually, because the fused process has to do the work of both source processes. In general, the larger the source processes, the larger the result process will be, and when we have many processes to fuse, the result will get progressively larger as we fuse more processes in. If, during compilation, the result process becomes too large such that the process does not fit in memory, then fusing in the next process will take longer, and code generation will take longer. When repeatedly fusing the pairs of processes in a network, we perform some simplifications between each fusion step. Since output channels can have multiple consumers, the result process pushes to all output channels used by either source process, even though all the consumers may have already been fused into the producer. We simplify the result program by removing any remaining output channels which are not read by the rest of the network, and replacing the corresponding `push` and `close` instructions with `jump` instructions. We then remove these `jump` instructions, as well as those introduced by the fusion algorithm, by inlining the destination of each `jump` instruction into its use-sites. This iterative simplification removes some unnecessary states and simplifies the process given to the next fusion step.

4.5.1 *Fusing a network*

As we shall see, when we fuse pairs of processes in a network, the order in which we fuse pairs can determine whether fusion succeeds. Rather than trying all possible orders, of which

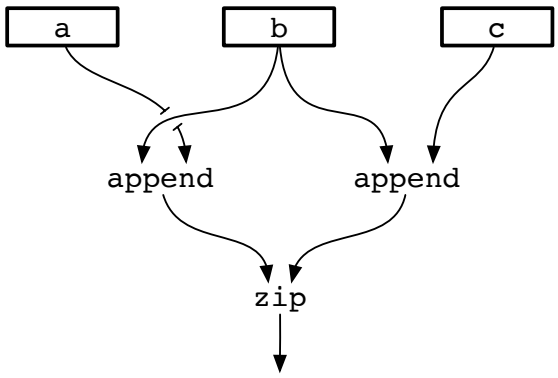


Figure 4.19: Dependency graph for `append2zip` example

there are many, we use a bottom-up heuristic to choose a fusion order. Figure 4.18 shows the heuristically chosen fusion order for the `priceAnalyses` example. The processes are nested inside boxes; each box denotes the result of fusing a pair of processes, and inner-most boxes are to be fused first. Each box is shaded to denote its nesting level, and the more deeply nested a box is, the darker its shade. In `priceOverTime`, we start by fusing the `correlation` process with its producer, `map`; we then fuse the resulting process with the `regression` process. In `priceOverMarket`, we also start by fusing the `correlation` process with `map`, then adding `regression`, and fusing in the `join` process. Finally, we fuse the result process for `priceOverTime` with the result process for `priceOverMarket`.

To demonstrate how fusion order can affect whether fusion succeeds, consider the following list program, which takes three input lists, appends them, and zips the appended lists together:

```
append2zip :: [a] → [a] → [a] → [(a,a)]
append2zip a b c =
  let ba = b ++ a
      bc = b ++ c
      z  = zip ba bc
  in  z
```

We use the more convenient syntax for list programs rather than the process network syntax introduced earlier, but in the discussion we interpret this program as a process network. In the process network interpretation, each list combinator corresponds to a process, and each list corresponds to a stream. The dependency graph for the corresponding process network is shown in Figure 4.19.

The append2zip program appends the input streams, then pairs together the elements in both appended streams. The result of the two append processes, ba and bc, both contain the elements from b stream, followed by the elements of the second append argument; stream a or stream c respectively. These two streams, ba and bc, when paired together, will result in each element of the b stream paired with itself, followed by elements of the two other streams paired together.

Figure 4.20 shows an example execution of append2zip, displayed as a sequence diagram. In this diagram, we omit the drop and pull internal messages for all processes, and focus instead on the communication between processes. In the definition of feeding for process networks, the (FeedEnvPush) rule takes values from external input streams and injects them into the process network as push messages. We visualise feeding as the input stream itself pushing elements to its consumers, just as if it were a separate process repeatedly pushing the elements. Input stream a has elements $[1, 2]$, input stream b has elements $[3, 4]$, and input stream c has elements $[5, 6]$. Input stream b has multiple consumers, so its elements are pushed to both consumers at the same time.

The execution has three sections. In the first section, all the values from the b stream are pushed to both append processes, then paired together. In the second section, execution alternates between the other streams, a and c, with one value from each. In the third section, the remaining input streams are closed, causing the consumers to also close their output streams.

The bottom-most consumer process, zip, executes by alternately pulling from each of the append processes. The order in which a process pulls from its inputs is called its *access pattern*. Each append process can only push when the zip process' buffer for that channel is empty: append must wait for zip to read the most recent element before pushing a new element. When each append process is waiting, its producer—the input stream—must also wait before pushing the next element. This waiting propagates the zip process' access pattern upwards through the append processes and to the input streams.

This example contains three processes. The fusion algorithm is not commutative or associative, so we could perform fusion in twelve different orders. Of these twelve orders, there are two main categories, distinguished by whether we start by fusing the append processes with each other, or start by fusing the zip process with one of the append processes.

If we fuse the two append processes together first, we interleave their instructions without considering the access pattern of the zip process. There are many ways to interleave the two processes; one possibility is that the fused process reads all of the shared prefix from stream b, then all of stream a, then all of stream c. For the shared prefix, this interleaving alternates

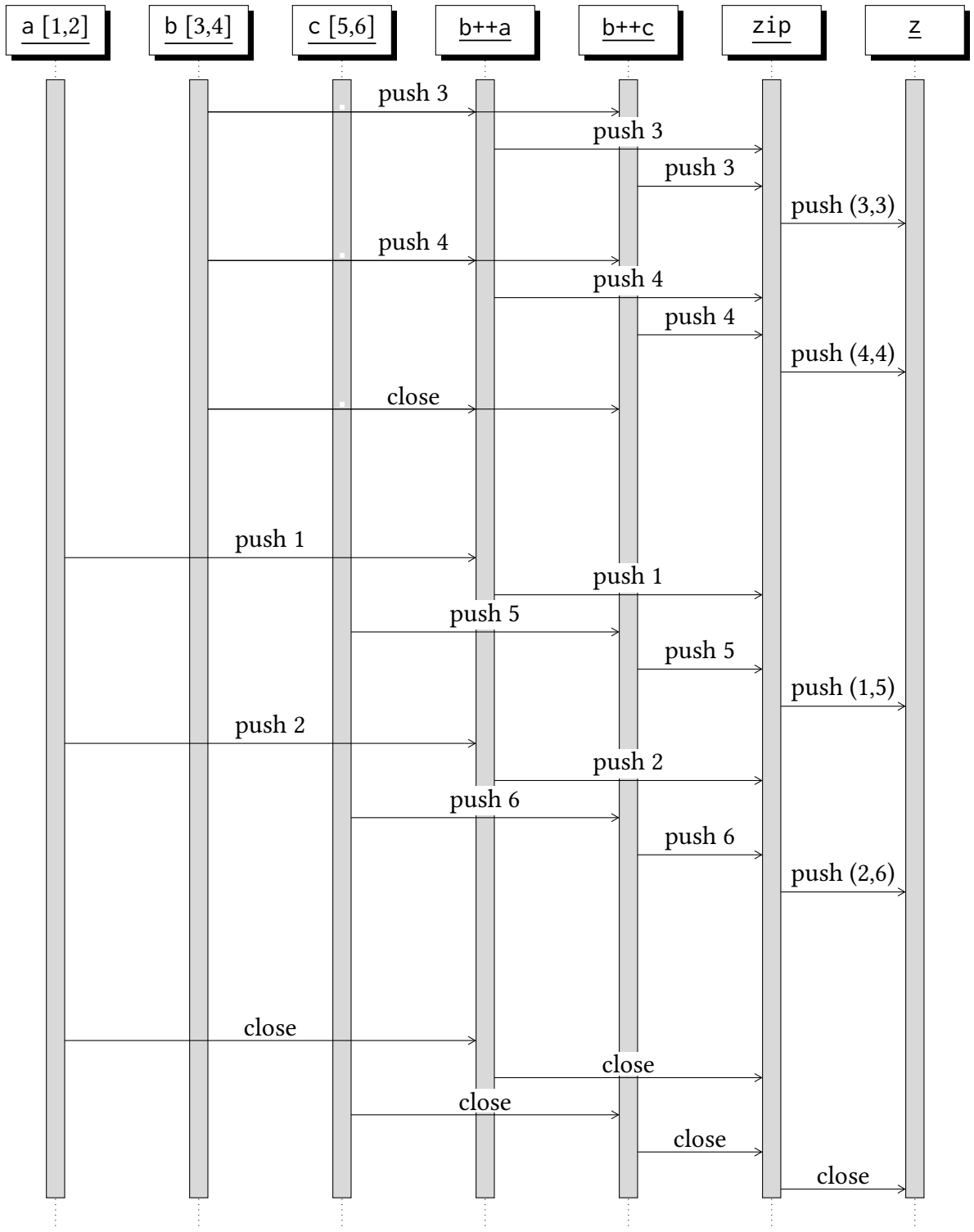


Figure 4.20: Sequence diagram of execution of `append2zip`

between pushing to streams ba and bc. After the shared prefix, this interleaving pushes the rest of the stream ba, then pushes the rest of the stream bc. When we try to fuse the `zip` process with the fused append processes with this interleaving, we get stuck. The `zip` process needs to alternate between its inputs, which works for the shared prefix, but not for the remainder. By fusing the two append processes together first, we risk choosing an interleaving that works for the two append processes on their own, but does not take into account the access pattern of the `zip` process.

Fusion does succeed if we fuse the `zip` process with one of the append processes first, then fuse with the other append process. The consumer, `zip`, must dictate the order in which the append processes push; fusing the `zip` process first gives it this control. We start from the consumer and fuse them upwards with their producers, because this allows the consumer to impose its access pattern on the producers.

To fuse an arbitrary process network, we consider a restricted view of the dependency graph, ignoring the overall inputs and outputs of the network. We start at the bottom of the dependency graph, finding the *sink* processes, or those with no output edges. These sink processes are the bottom-most consumers which, like `zip` in our `append2zip` example, dictate the access pattern on their inputs. For each sink process, we find its parents and fuse the sink process with its parents. When the sink process has multiple parents, we need to choose which parent to fuse with first. In the `append2zip` example, we can fuse the `zip` process with its append parents in any order. In general, one parent may consume the other parent's output; in which case we first fuse with the consumer parent. This order allows the consuming parent to impose its access pattern upon the producing parent. We repeatedly fuse each sink process with its closest parent until there are no more parents.

After fusing each sink process with all its ancestors, there may remain multiple processes. This only occurs if the remaining processes do not share ancestors. The remaining processes also cannot share descendents, since if they had descendents they would not be sinks. This means the processes are completely separate and could be executed separately, in any order or even in parallel. Having unconnected processes in the same process network is a degenerate case, as it could be represented as multiple process networks. We execute the remaining unconnected processes concurrently, as a dynamically-scheduled concurrent process network.

The fusion algorithm for pairs of processes fails and does not produce a result process when two processes have conflicting access patterns on their shared inputs. As the access patterns are determined statically, apparent conflicts may never occur at runtime; we instead make a static approximation. In the implementation, we fuse as many pairs of processes together as

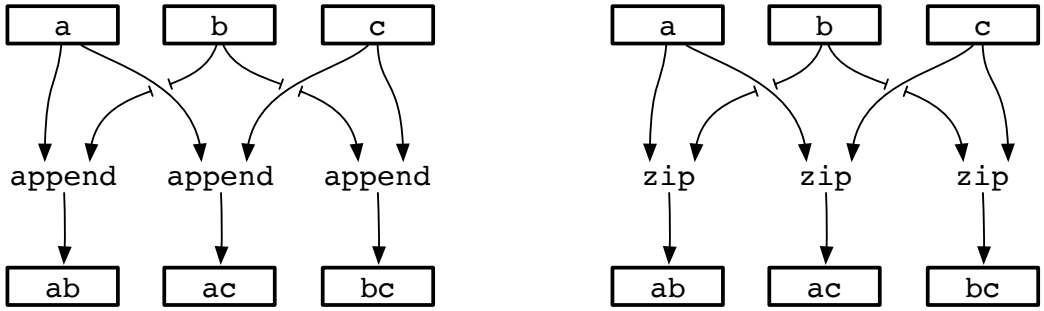


Figure 4.21: Dependency graphs for `append3` and `zip3` examples

possible. If at any point we encounter a pair of processes which we cannot fuse together, we display a compile-time warning telling the programmer that the network cannot be completely fused. The remaining partially-fused processes are executed concurrently.

Unfortunately, the above heuristic cannot always choose the correct fusion ordering. It is not possible, in general, to choose the correct ordering based on the dependency graph alone. Consider the following list program, `append3`, which appends three input lists in various orders, producing three output lists. As with the `append2zip` example, we present the example as a list program for syntactic convenience, while we interpret it as a process network:

```
append3 :: [a] → [a] → [a] → ([a],[a],[a])
append3 a b c =
  let ab = a ++ b
      ac = a ++ c
      bc = b ++ c
  in  (ab, ac, bc)
```

This process network can be executed with no buffering. First, read all of the `a` input stream, then read the `b` stream, then read the `c` stream. There is no single consumer in this example which imposes its access pattern on its producers, so our heuristic fails. This process network can only be fused if the process that produces `ab` and the process that produces `bc` are first fused together. If we fused `ab` and `ac` together first, the fusion algorithm would make an arbitrary decision of whether to read the `b` stream before, after, or interleaved with the `c` stream. The heuristic described will not necessarily choose the right order.

Looking at the dependency graph alone, it is impossible to tell which is the right order for fusion. If we take the `append3` example and replace the append processes with `zip` processes, the dependency graph remains the same, but either fusion order would work. Figure 4.21 shows the process networks of `append3` and `append3` replaced with `zip` processes. Whether

or not a particular fusion order works depends on the access pattern of the processes, which is not shown in the dependency graph.

We propose to solve this in future work by modifying the fusion algorithm to be commutative and associative. These properties would allow us to apply fusion in any order, knowing that all orders produce the same result. We discuss this enhancement further in Section 8.1. In the meantime, we fall back to dynamic scheduling and execute the partially-fused process networks concurrently.

4.6 PROOFS

Our fusion system is formalised in Coq[1], and we have proved soundness of *fusePair*: if the fused result process produces a particular sequence of values on its output channels, then this is one of the possible sequences that would be produced by the two source processes. Note that due to the non-determinism of process execution, the converse is not true for all source processes: just because the two concurrent processes can produce a particular output sequence does not mean the fused result process will as well — the fused result process uses only one of the many possible orders. However, because the result of evaluating a Kahn process network to completion *is* deterministic, we should be able to prove that, if fusion succeeds, the result process produces the same overall result despite using potentially different interleavings. The proof of result determinism is left to future work.

The proof of soundness is stated as follows:

```
Theorem Soundness (P1 : Program L1 C V1) (P2  : Program L2 C V2)
                  (ss : Streams)         (h   : Heap)
                  (l1 : L1)              (is1 : InputStates)
                  (l2 : L2)              (is2 : InputStates)
  :  EvalBs (fuse P1 P2) ss h (LX l1 l2 is1 is2)
  → EvalOriginal Var1 P1 P2 is1 ss h l1
  ∧ EvalOriginal Var2 P2 P1 is2 ss h l2.
```

The `Soundness` theorem uses `EvalBs` to evaluate the fused program, and `EvalOriginal` ensures that the original program evaluates with that program's subset of the result heap, using `Var1` and `Var2` to extract the variables. The `Streams` type corresponds to the channel value map used to accumulate stream elements while feeding a process network (Figure 4.9), and the `Heap` type corresponds to the value store used while advancing a single process (Figure 4.8).

[1] https://github.com/amosr/papers/tree/master/2017mergingmerges/proof

To aid mechanisation, the Coq formalisation has some small differences from the system presented earlier in this thesis. Firstly, the Coq formalisation uses a separate `update` instruction to modify variables in the local heap, rather than attaching heap updates to the *Next* label of every instruction. Performing this desugaring makes the low-level lemmas easier to prove, but we find attaching the updates to each instruction makes for an easier exposition. Having a separate `update` instruction causes the fusion definition to be slightly more complicated, as two output instructions must be emitted when performing a push or pull followed by an update. This difference is fairly minor.

Secondly, the formalisation only implements sequential evaluation for a single process, rather than non-deterministic evaluation for whole process networks. Instead, we sequentially evaluate each source process independently, and compare the output values to the ones produced by sequential evaluation of the fused result process. To allow both input processes to sequentially evaluate to the same collected stream values, the sequential evaluation includes the concept of external events, such as another process pushing to a stream. This is sufficient for our purposes because we are mainly interested in the value correctness of the fused program, rather than making a statement about the possible execution orders of the source processes when run concurrently.

Like the earlier presentation, each program has a mapping from labels to instructions. We also associate each label with an invariant, which is expressed as a predicate of the evaluation state. The invariant for the initial label must be true for the initial evaluation state. Whenever the program takes an evaluation step, assuming the invariant for the original label was satisfied at the start of the step, the invariant for the result label must be satisfied by the updated evaluation state. Thus, each label's invariant serves the purpose of both precondition for the corresponding instruction, and postcondition for predecessor instructions.

For the fused result process, we generate a new invariant that all evaluations must satisfy. The invariant is similar to the definition of `Soundness` above, and defines the simulation relation between evaluations of the result process and evaluations of the original source processes. Each result process label is defined as the pair of source process labels, along with the static input states. For a particular result process label, the invariant states that, assuming the result process evaluates to this label with a particular value heap and channel value map, then both the source processes evaluate to the corresponding label with their corresponding subset of the heap, and a similar channel values map. The channel values map for the source processes may differ slightly. If, for example, the first source process has pushed a new value to a shared stream, and the second source process has not yet pulled the new value, the new value will

appear in the channel value map for the first process, but not in the channel value map for the second process. In this example, the second source process has an input state of `pending`, which denotes that the most recent value has not yet been pulled, and should not be included in the evaluated channel value map for that source process.

Proving that the invariant is established initially is straightforward, as the channel value map is empty and the result process' initial heap contains both source processes' initial heaps. To prove that the invariant is maintained for a particular label in the result process, we perform case distinction on the instruction corresponding to the label, then, as necessary, perform case distinction on the original source process instructions. With the invariant established and maintained, we prove the `Soundness` theorem by performing induction over the evaluation relation, and showing that all evaluations of the result process respect the invariant. We believe that this proof gives sufficient confidence in the correctness of the presentation given earlier, despite the minor differences in formulation.

CHAPTER 5

EVALUATION OF PROCESS FUSION

Stream fusion is ultimately performed for practical reasons: we want the fused result program to run faster than the original unfused program. We also need to be sure that the fused program is not too large, to ensure that compilation and the fusion transform do not consume too much memory. This chapter shows runtime benchmark results for fused programs, and the result size of fused programs.

5.1 BENCHMARKS

The process fusion algorithm described in Chapter 4 is implemented in a library called *Folderol*[1]. Our implementation uses the topology of the entire process network to perform fusion, statically coordinating between all processes in the network. To fuse the entire process network at compile-time, Folderol uses Template Haskell, a form of metaprogramming.

Benchmarks are available at `https://github.com/amosr/folderol/tree/bench/bench`. As well as the "gold panning" example from Section 2.1, we present one spatial algorithm, two file-based benchmarks, and one audio signal processing benchmark. In the benchmarks, we compare Folderol variously against hand-fused implementations, the array library Vector (Leshchinskiy, 2008), as well as streaming libraries Conduit (Snoyman, 2011), Pipes (Gonzalez, 2012) and Streaming[2] (Thompson, 2015). We restrict our attention to Haskell libraries to isolate the cost of streaming from language implementation details. The concepts in the chosen libraries are applicable to other languages, and we expect other implementations to show comparable benchmark results. Strymonas (Kiselyov et al., 2017), which has OCaml and Scala implementations, uses a similar stream representation to Vector, but uses staged computation to perform fusion. Functional streams for Scala (Chiusano, 2013) uses a similar representation to Pipes and Conduit.

The Vector library provides high-performance boxed and unboxed arrays with pull-based shortcut fusion. It is the *de facto* standard for array programming in Haskell, and implements

[1] `https://github.com/amosr/folderol`

[2] We use this typography to distinguish Streaming the library from streaming the abstract concept.

```
data ConduitT i o m r =
    NeedInput  (i → ConduitT i o m r) (ConduitT i o m r)
  | HaveOutput (ConduitT i o m r) o
  | Done r
  | ConduitM   (m (ConduitT i o m r))
  | Leftover   (ConduitT i o m r) i

instance Monad m ⇒ Monad (ConduitT i o m)
```

Listing 5.1: Conduit datatypes

a shortcut fusion system called *stream fusion*, introduced by Coutts et al. (2007). Array operations are implemented by converting the input array to a stream, then performing the corresponding stream operation, and converting back to an array. The shortcut fusion rule removes the superfluous conversion when a stream is converted to an intermediate array and immediately back to a stream. Just as pull streams only support a single consumer, this rule can only remove intermediate arrays which have a single consumer. If the same intermediate array is used multiple times, it acts as a *fusion barrier*, forcing the stream to be manifested into a real, memory-backed array. We discuss fusion barriers further in Chapter 7. In practice, Vector provides no guarantees about whether fusion will occur, and the easiest way to tell whether fusion has occurred is to look at the generated code to count the loops and arrays. We could benchmark the program to see whether it is "fast enough", but if we do not have an optimal baseline to compare against, it is hard to know how fast the program *should* be. After inspecting the generated code, if we discover that fusion has not occurred, it may be necessary to rewrite the program and hand-fuse it.

The first two Haskell streaming libraries, Conduit and Pipes, both have limited APIs that ensure that any computation can be run with bounded buffers. These streaming libraries do not naturally support multiple queries: when a stream is shared among multiple consumers, part of the program must be hand-fused, or somehow rewritten as a straight-line computation. These libraries also have a monadic interface, which allows the structure of the dataflow graph to depend on the values. This expressiveness has a price: if the dataflow graph can change dynamically, we cannot statically fuse it.

Listing 5.1 shows a simplified version of the core datatype for Conduit. The `ConduitT` type defines a stream transformer that consumes elements of type `i` and produces elements of type `o`. The transformer may perform effects in the `m` monad, and returns a final result value of type `r`. The `NeedInput` constructor pulls from the input and takes two arguments: a function which

```
data Stream f m r =
    Step   (f (Stream f m r))
  | Effect (m (Stream f m r))
  | Return r
instance (Functor f, Monad m) ⇒ Monad (Stream f m)

data Of a b = a :> b
instance Functor (Of a)

store :: Monad m ⇒
      (Stream (Of a) (Stream (Of a) m) r → t) →
      Stream (Of a) m r → t

sum  :: (Num a, Monad m) ⇒
      Stream (Of a) m r → m (Of a r)
```

Listing 5.2: Streaming library datatypes

takes the input element and returns a new stream transformer, and a stream transformer to use when the input stream is finished. The `ConduitM` constructor performs a monadic effect, which returns a new stream transformer for the rest of the stream. Because the data structure describing remainder of the stream is nested inside a closure that performs a monadic computation, the compiler may not be able to statically reason about the remainder of the stream. The constructors and closures for the remainder of the stream will be allocated at runtime, instead of being statically optimised away. To reduce this overhead, Conduit also implements a fusion system based on Vector's stream fusion (Coutts et al., 2007). In stream fusion, the stream is decomposed into a static, non-recursive step function and a dynamic state. The static, non-recursive step function is simpler to inline and optimise than the recursive structure. Many of the combinators are implemented in terms of both stream fusion as well as the Conduit implementation. Rewrite rules are used to replace the Conduit implementation with the stream fusion implementation where possible. Some Conduit operations, such as monadic bind, cannot be efficiently implemented in terms of stream fusion, because the structure of the stream can dynamically depend on the bound value.

The Pipes library uses a recursive, monadic data structure to represent stream transformers. This representation is similar to the representation used by Conduit, and involves similar runtime overhead. Instead of using stream fusion, Pipes has an extensive set of rewrite rules to statically optimise particular sequences of operations.

The Streaming library is a monadic streaming library, similar to Conduit and Pipes. Listing 5.2 shows the core datatypes for Streaming. The `Stream` type defines a stream producer parameterised by a functor `f`, a monad `m`, and a result type `r`. The functor for a stream producer is often instantiated to (`Of a`), where `a` is the type of the producer's stream elements. When the functor is instantiated to (`Of a`), the `Step` constructor for `Stream` will contain the produced stream element of type `a`, as well as the remaining stream. The `Effect` constructor, like the `ConduitM` constructor in Conduit, performs an effect and returns the remaining stream. The `Return` constructor signals the end of the stream, and returns the result `r`. Streaming can execute multiple queries by explicitly duplicating streams, similar to polarised streams (Section 2.4). To duplicate the stream, the `store` function takes a computation to perform on the copy of the stream. This operation is analogous to the `dup_ioi` combinator, which duplicates a pull stream into a push stream and returns a new pull stream; however, the implementation encodes stream polarity with a monad transformer stack rather than directly using pull and push streams. The computation passed to `store` is a function that takes a `Stream` representing the duplicated stream, with the original stream nested inside the monad transformer stack. We can sum the elements of the duplicated stream while passing through the original elements like so:

```
store sum :: Num a ⇒ Stream (Of a) m r → Stream (Of a) m (Of a r)
```

The input stream has the result type `r`, while the modified stream contains the same elements and has the result type (`Of a r`), which is effectively a pair of the sum and the original result. By putting the original stream in the duplicated stream's monad transformer stack, the original stream's elements are encoded as effects to be performed by the sum computation, while it pulls from the duplicated stream. Unfortunately, because each stream element involves constructing the `Stream` recursive data structure with a stack of monad transformers, the corresponding runtime allocations and closures are not always statically optimised away.

The benchmark results presented in this chapter were all run on a MacBook Pro, with a 2GHz Intel Core i7, and 16GB of RAM. The operating system is OS X El Capitan with Glasgow Haskell Compiler (GHC) 8.0.2. To run the benchmarks, we used the Criterion[3] library. Criterion offers a fairly reliable way to evaluate runtime performance, as it runs each benchmark multiple times to compute the mean runtime, and warns if the variance is too high or if there are outliers. It can also collect statistics on allocations, which can be a more stable

[3] `http://hackage.haskell.org/package/criterion`

```
priceAnalysesFolderol :: (FilePath,FilePath) → IO (Double,Double)
priceAnalysesFolderol (fpStock, fpMarket) = do
  (pot,(pom,())) ← scalarIO $ λsnkPOT → scalarIO $ λsnkPOM →
    $$(fuse $ do
      stock  ← source [|sourceRecords fpStock|]
      market ← source [|sourceRecords fpMarket|]
      pot' ← priceOverTime   stock
      pom' ← priceOverMarket stock market
      sink pot' [|snkPOT|]
      sink pom' [|snkPOM|])
  return (pot,pom)

priceOverTime stock = do
  tp ← map [|λs → (daysSinceEpoch (time s), price s)|] stock
  Stats.regressionCorrelation tp

priceOverMarket stock market = do
  j  ← joinBy [|λs m → time s `compare` time m|] stock market
  pp ← map    [|λ(s,m) → (price s, price m)|] j
  Stats.regressionCorrelation pp
```

Listing 5.3: Folderol implementation of `priceAnalyses`

performance indicator than runtime, as it is less affected by the underlying operating system's scheduling decisions.

We have implemented each benchmark program multiple times across the different backends. These different implementations generally follow the same structure. All the implementations are available in Appendix A; in this chapter, we focus on the Folderol versions.

5.1.1 *Gold panning*

Our first benchmark is the `priceAnalyses` example from Section 2.1. This example computes statistical analyses of a particular stock's price over time, as well as how the stock compares to a market index.

Listing 5.3 shows the Folderol implementation of `priceAnalyses`. The calls to `scalarIO` indicate that the query returns two scalar variables that we wish to capture. The `fuse` function converts the process network into executable code. The dollar-sign syntax around `fuse` denotes a Template Haskell splice, which means that the call to `fuse` is evaluated at compile-time, and returns code to execute at runtime. The `fuse` function takes a process network constructed at

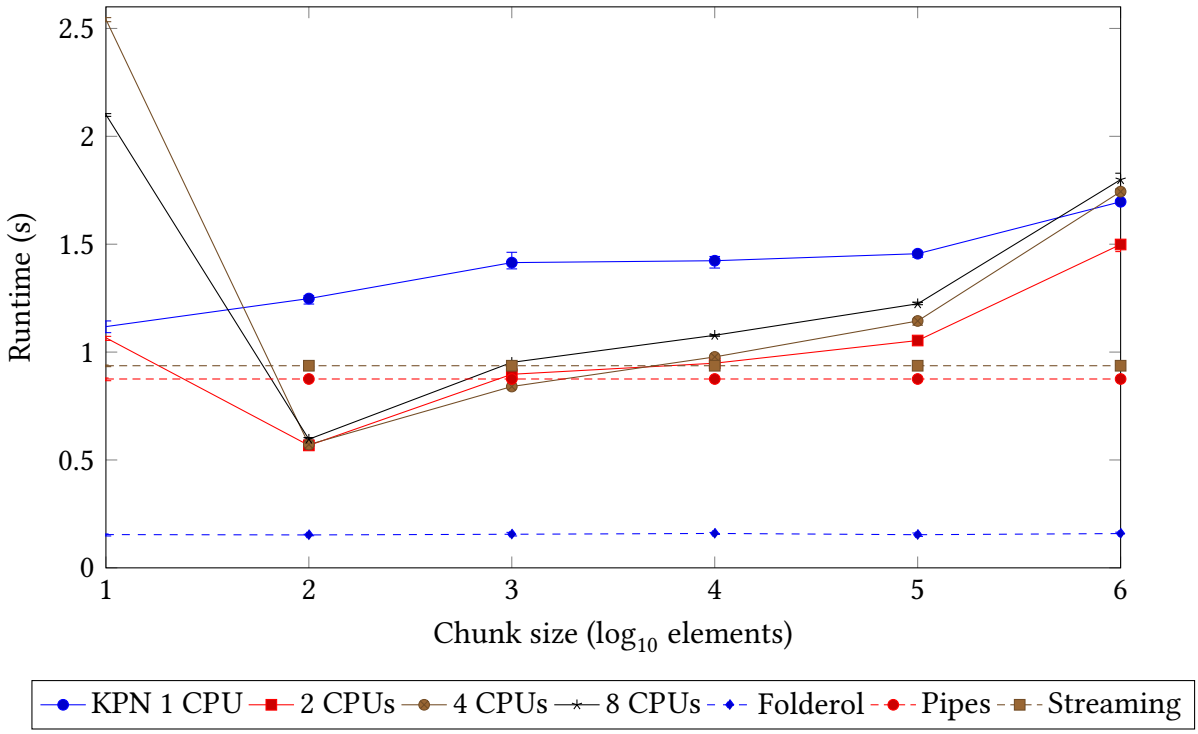


Figure 5.1: Runtime performance for `priceAnalyses` queries

compile-time, fuses the network into a single process, and generates code for the resulting process. In the call to `source`, the piped brackets around the argument `[|sourceRecords fpStock|]` indicate a Template Haskell quasiquote; this is used to delay execution of code from compile-time to runtime.

We use the same Folderol implementation to benchmark against concurrent execution of a Kahn process network. We disable fusion by replacing the call to `fuse` with a function called `fuseWith`, which takes a set of fusion and code generation options. These options dictate whether to perform fusion, whether to automatically insert communication channels between each process, and if so what chunk size to use for channels. After inserting the communication channels, the concurrent version uses the same code generation backend as the fused version.

For Pipes, which does not naturally support multiple queries, we implement a two-pass version which computes `priceOverTime` and `priceOverMarket` in separate loops over the input. The implementation is available in Listing A.1.

For Streaming, we duplicate the stream explicitly to compute `priceOverTime` as a push stream. The implementation is available in Listing A.2.

Figure 5.1 shows the results of benchmarking the different implementations of `priceAnalyses` over 10^6 elements. We ran with different sizes of data ranging from 10^3 to 10^8 but do not show

these additional values; the results for this size are representative of the relationship between the different implementations. We compare the concurrent execution of Kahn process networks (KPN) with one, two, four, and eight processors. For the concurrent executions, we vary the chunk size along the x axis, to try to find the best trade-off between memory usage and communication overhead. For the sequential implementations — Folderol, Pipes, and Streaming— the inter-process chunk size does not affect execution.

Interpreting the results in Figure 5.1, we see that the fused Folderol implementation is the fastest. For concurrent execution of the Kahn process network implementation with multiple processors, a chunk size of one hundred appears to be ideal. For this example with only a few queries, there is a significant improvement between one and two processors, but little further improvement as more processors are added. With the ideal chunk size, the concurrent execution of the Kahn process network is 1.5 times faster than the Pipes implementation, and 3.7 times slower than the fused version. The Folderol version is completely statically fused, and allocates very few intermediate data structures, while the concurrent Kahn process network is dynamically scheduled and involves some channel communication overhead. The Pipes and Streaming implementations cannot statically fuse all operations, so allocate more intermediate data structures. The Pipes implementation also performs two passes over the input, and must read the file twice.

5.1.2 *Quickhull*

Quickhull is a divide-and-conquer spatial algorithm to find the smallest convex hull containing all points. At its core is the `filterMax` operation which takes a line and an array of points, and finds the farthest point above the line, as well as all points above the line.

Listing 5.4 shows the Folderol implementation of `filterMax`. The Folderol implementation starts by constructing a sink for the maximum point to be pushed into (`snkMaxim`), and a sink for the vector of points above the line (`snkAbove`). As we know the output vector of points is not going to be any longer than the input vector, we use `vectorSize` as a size hint to specify the upper bound of the vector's size. The Template Haskell splice calls `fuse` to convert the process network into executable code. The process network starts by converting the input vector `pts` to a stream `ins`. We use `source` to create an input stream, which takes a Template Haskell expression denoting how to construct a source at runtime. We then annotate each point of `ins` with the distance between each point and the line with `map`, calling this stream `annot`. The `annot` stream is filtered to only those above the line (`above`), then the annotations thrown away

```
filterMaxFolderol :: Line → Vector Point → IO (Point, Vector Point)
filterMaxFolderol l ps = do
  (maxim,(above,())) ← scalar          $ λsnkMaxim →
                       vectorSize   ps $ λsnkAbove →
   $$(fuse $ do
      ins    ← source [|sourceOfVector ps      |]
      annot  ← map    [|λp → (p, distance p l)|] ins
      above  ← filter [|λ(_,d) → d > 0        |] annot
      above' ← map    [|fst                   |] above
      maxim  ← maxBy  [|compare `on` snd      |] annot
      sink maxim      [|snkMaxim              |]
      sink above'     [|snkAbove              |])
  return (fst maxim, above)
```

Listing 5.4: Folderol implementation of `filterMax`

```
filterMaxVectorShare l ps
 = let annot = map (λp → (p, distance p l)) ps
       point = fst
             $ maximumBy (compare `on` snd) annot
       above = map fst
             $ filter ((>0) ∘ snd) annot
   in return (point, above)
```

Listing 5.5: Vector / share implementation of filterMax

(`above'`). The maximum is computed by comparing the second half of each annotated point — the distance — and stored in `maxim`. Finally, `maxim` is pushed into the scalar output sink, and all `above'` points are pushed into the vector output sink.

In the Folderol implementation, the `annot` stream is used twice. If we rewrite this to use Vector, as in Listing 5.5, the corresponding `annot` vector cannot be fused with both of its consumers, and the program requires multiple loops. We call this the 'shared distances' version, as the `annot` vector containing the distances is made manifest and re-used by the two loops computing the maximum and the filter.

Instead of constructing `annot` as a manifest vector and sharing it among the two consumer loops, we can also recompute the distances for each consumer. The recomputed distances implementation is available in Listing A.6. To recompute the vectors, we modify the shared version, duplicating the binding for the `annot` vector to `annot1` and `annot2`. The maximum now uses `annot1`, and filter uses `annot2`. This way, both maps to compute the distance can be fused into their consumers.

```
filterMaxConduitTwoPass l ps = do
  maxim ← runConduit cmaxim
  above ← runConduit cabove
  return (fst maxim, above)
 where
  cabove =
    sourceVector ps               .|
    map (λp → (p, distance p l)) .|
    filter ((>0) ∘ snd)           .|
    map fst                       .|
    sinkVectorSize ps

  cmaxim =
    sourceVector ps               .|
    map (λp → (p, distance p l)) .|
    maximumBy (compare `on` snd)
```

Listing 5.6: Conduit two-pass implementation of `filterMax`

In the results below, recomputing the distances is faster and allocates less than the shared distances version. For this benchmark we used only two-dimensional points, but it is possible that at higher dimensions the cost of recomputing distances may outweigh the cost of allocation. The choice to recompute the distances requires intimate knowledge of how shortcut fusion works and might be surprising to the naive user: duplicating work does not usually *improve* performance.

For Conduit, we also have two versions: the first uses two passes over the data (Listing 5.6), while the second is hand-fused into a single pass (Listing 5.7). The two-pass version defines two 'conduits', or stream computations; `cabove` for the filtered vector and `cmaxim` for the maximum. Computation `cabove` converts the vector to a conduit with `sourceVector`, then annotates with the distances, filters according to the distances, removes the annotations, and converts back to a vector. As with Folderol, we use size hints when converting back to a vector to remove any overhead with growing and copying the vector. Computation `cmaxim` also converts the vector to a conduit, then annotates with the distances, and computes the maximum. The hand-fused Conduit version is more complicated, and loses some of the abstraction benefits from using a high-level streaming library.

For Pipes, we only have a hand-fused version (Listing A.7), which follows much the same structure as the Conduit hand-fused version.

```
filterMaxConduitOnePass l ps = do
  r      ← MUnbox.unsafeNew (Unbox.length ps)
  (a,ix) ← runConduit $ both r
  r'     ← Unbox.unsafeFreeze $ MUnbox.unsafeSlice 0 ix r
  return (a, r')
 where
  both r =
    sourceVector ps               .|
    map (λp → (p, distance p l)) .|
    filterAndMax r 0 (0,0) (-1/0)

  filterAndMax !r !ix (!x,!y) !d1 = do
    e ← await
    case e of
     Just (!p2,!d2) → do
      let (!p',!d') = if d1 > d2 then ((x,y),d1) else (p2,d2)
      case d2 > 0 of
       True → do
        MUnbox.unsafeWrite r ix p2
        filterAndMax r (ix+1) p' d'
       False → do
        filterAndMax r ix p' d'
     Nothing → do
      return ((x,y), ix)
```

Listing 5.7: Conduit one-pass (hand-fused) implementation of `filterMax`

```
filterMaxStreaming l ps = do
  (vec,pt :> ()) ← sinkVectorSize ps
            $ map fst
            $ filter (λ(_,d) → d ≠ 0)
            $ store maximumBy (compare `on` snd)
            $ map (λp → (p, distance p l))
            $ sourceVector ps
  return (pt, vec)
```

Listing 5.8: Streaming implementation of `filterMax`

		Runtime (s)	Runtime (%)	Allocation (b)	Allocation (%)
Folderol		0.21s	100%	1.2e9	100%
Hand		0.14s	66%	4.8e8	40%
Vector	store	0.40s	190%	1.8e9	150%
	recompute	0.34s	161%	8.0e8	66%
Conduit	2-pass	10.0s	4,762%	6.2e10	5,167%
	hand	7.9s	3,762%	4.4e10	3,667%
Pipes	hand	4.9s	1,876%	3.9e10	3,250%
Streaming		4.4s	2,095%	3.0e10	2,500%

Table 5.1: Quickhull benchmark results

Finally, the STREAMING version (Listing 5.8) executes in a single pass, with the stream explicitly duplicated into both consumers. We start by converting the vector to a stream with `sourceVector`, and then use `map` to annotate each point with the distance from the line. To duplicate the stream we use the `store` combinator, which takes a computation to perform on the copy of the stream — in this case computing the maximum with `maximumBy`. The original stream is also passed through, which is filtered and then the annotations discarded.

Table 5.1 shows the runtimes for Quickhull over 10^7 points, which corresponds to around 150MB in memory. The hand-fused version is the fastest, followed by Folderol. The Folderol version performs a single loop, but the generated code stores the result vector as a boxed object, and reboxes the vector for every write to the vector. Our code generation generates Haskell source code and relies on the Glasgow Haskell Compiler's constructor specialisation optimisation (Peyton Jones, 2007) to unbox loop variables. Most loop variables are unboxed by constructor specialisation, but constructor specialisation uses heuristics to determine which constructors to remove, and in this case does not unbox the result vector. In the future, we may be able to improve the code generation by performing the unboxing ourselves for certain loop variables with a statically known data constructor.

The Vector versions are somewhat slower than the hand-fused and Folderol versions because they require two iterations over the data, but they still generate high-quality loops. Intimate knowledge of shortcut fusion was required to find 'recompute', the faster of the two vector benchmarks, and there is no indication in the source program, or from the compiler, that the two could not be fused together. The streaming libraries are significantly slower, spending

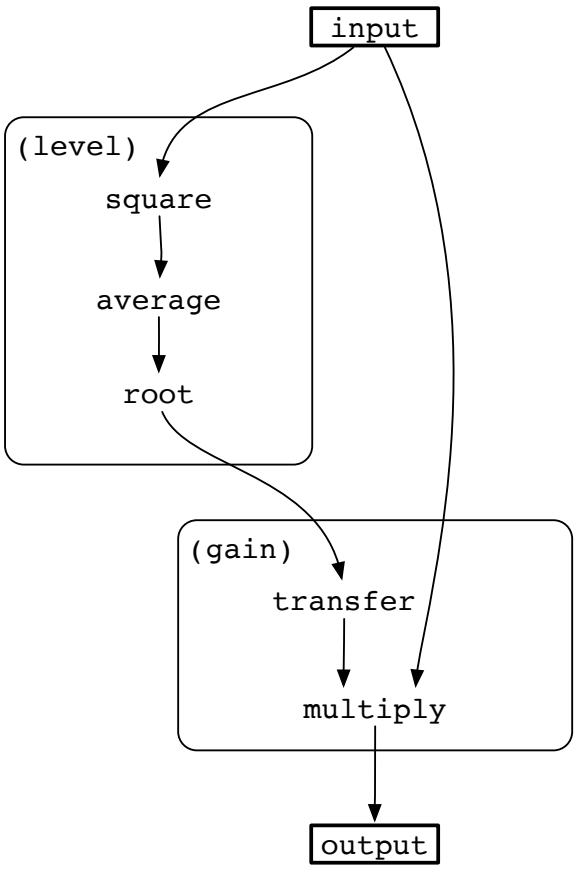


Figure 5.2: Dependency graph for audio compressor

most of the time allocating closures and forcing thunks. On the plus side, the limited APIs and linearity constraints for Conduit and Pipes made it more obvious that they would not be fused into a single loop, but even with partial hand-fusion, are still significantly slower. The Streaming program allows explicit sharing, and was the simplest of the three streaming libraries, requiring no hand fusion. Surprisingly, the simpler Streaming program is faster than the hand-fused Conduit and Pipes programs, but is still significantly slower than Folderol.

These streaming libraries are not usually used for array computations because of this overhead. In practice, one tends to 'chunk' the data so that each element is an array of multiple elements: this reduces overhead paid for streaming. This chunking must be implemented manually and complicates the program further, reducing the level of abstraction the programmer can work at. Even with chunking, we may reduce the streaming overhead from once per element to once per thousand elements, but we can never remove it entirely. With Folderol there is no streaming overhead to reduce because it is statically fused away.

5.1.3 *Dynamic range compression*

Dynamic range compression is an audio signal processing algorithm to reduce the volume of loud sounds, leaving quiet sounds unchanged. To implement this, we take an audio signal, and at each point approximate the signal's current volume with a rolling average. We pass the current volume to a *transfer function* to compute the *gain multiplier*, which denotes how much to adjust the volume by. If the volume of the input signal is not above the volume threshold, the transfer function sets the gain multiplier to one, indicating no change to the input signal. If the volume is louder than the threshold, the transfer function decreases the volume by setting the gain multiplier to a value below one.

Figure 5.2 shows the dependency graph for the audio compressor. The input stream is used twice, once by the level detector subgraph, and again by the gain subgraph. The level detector subgraph approximates the signal volume by computing the root mean square with an exponential moving average. The gain subgraph uses the output of the level detector to compute the gain multiplier, and then uses this to adjust the volume of the input signal.

The Folderol implementation is shown in Listing 5.9. The stream transformers `square`, `root` and `transfer` are computed by applying a single function to each stream element. To compute the moving average, we use `postscanl`, which is like a fold over the data except it returns the stream of accumulators rather than just the final result. The difference between `postscanl` and regular `scanl` is that the output stream for `scanl` contains the original zero accumulator for the fold, which `postscanl` omits.

The Vector implementation (Listing A.8) is similar to the Folderol implementation, except it does not require the Template Haskell fusion directives or explicit conversion between arrays and streams. Vector's shortcut fusion is able to fuse this program into a single loop, except the two occurrences of the input vector mean that the vector will be indexed twice.

Table 5.2 shows the runtimes for the dynamic range compressor over 10^8 samples, which corresponds to around 750MB in memory. The runtime difference between the hand-fused implementation and the Folderol implementation is very slight. The Vector implementation is slower because its generated loop indexes the input array twice.

5.1.4 *Dynamic range compression with low-pass*

Before applying dynamic range compression to an audio signal, we may wish to perform some pre-processing on it. For this benchmark, we want to apply a low-pass filter to the input signal.

```
compressor :: Vector Double → IO (Vector Double)
compressor vecIns = do
  (vecOut,()) ← vectorSize vecIns $ λsnkOut → do
    $$(fuse $ do
        input   ← source    [|sourceOfVector vecIns|]
        squares ← square    input
        means   ← average   squares
        roots   ← root      means
        gains   ← transfer  roots
        output  ← multiply  input    gains
        sink output         [|snkOut|])
  return vecOut

-- Stream transformers
square   = map       [|λx → x * x|]
average  = postscanl [|average'   |] [|0|]
root     = map       [|sqrt       |]
transfer = map       [|transfer'  |]
multiply = zipWith   [|(*)        |]

-- Worker functions
average' acc sample
 = acc * 0.9 + sample * 0.1

transfer' mean
 | mean > 1  = 1 / mean
 | otherwise = 1
```

Listing 5.9: Folderol implementation of `compressor`

	Runtime (s)	Runtime (%)	Allocation (b)	Allocation (%)
Folderol	0.93s	100%	8.0e8	100%
Hand	0.98s	105%	8.0e8	100%
Vector	1.78s	191%	8.0e8	100%

Table 5.2: Compressor benchmark results

```
compressorLop :: Vector Double → IO (Vector Double)
compressorLop vecIns = do
  (vecOut,()) ← vectorSize vecIns $ λsnkOut → do
    $$(fuse $ do
        input   ← source    [|sourceOfVector vecIns|]
        lopass  ← postscanl [|lop20k|] [|0|] input
        squares ← square    lopass
        means   ← average   squares
        roots   ← root      means
        gains   ← transfer  roots
        output  ← multiply  input    gains
        sink out            [|snkOut|])
  return vecOut
```

Listing 5.10: Folderol implementation of `compressor` with low-pass

	Runtime (s)	Runtime (%)	Allocation (b)	Allocation (%)
Folderol	1.16s	100%	8.0e8	100%
Hand	1.18s	101%	8.0e8	100%
Vector	2.21s	190%	1.6e9	200%

Table 5.3: Compressor with low-pass benchmark results

The Folderol implementation of a dynamic range compressor with a low-pass is shown in Listing 5.10. The low-pass filter is itself a kind of moving average, and is defined in much the same way as the `average` stream transformer. We use `postscanl` to create a low-pass filter to remove frequencies above 20kHz, and call the result stream `lopass`. The `lopass` stream is then used as the input to the level detector subgraph and the gain subgraph.

Table 5.3 shows the runtimes for compressor with low-pass. For the Vector implementation in the previous example, the two occurrences of the input vector meant that the input array was indexed twice. However, in the modified version with low-pass (Listing A.9), it is the `lopass` vector which is used twice, rather than the input vector. Using the `lopass` vector twice means that it must be reified into a manifest vector before it can be reused. The Vector version then has two loops, as well as an extra manifest array.

In a real-time audio compressor, the input audio signal would be a potentially infinite stream, rather than an in-memory vector. Folderol could be extended to operate over infinite streams, and our original paper on process fusion (Robinson and Lippmeier, 2017) described a process fusion algorithm where all streams are infinite. The Vector implementation cannot

```
append2 :: FilePath → FilePath → FilePath → IO Int
append2 fpIn1 fpIn2 fpOut = do
  (count,()) ← scalarIO $ λsnkCount →
    $$(fuse $ do
      in1   ← source [|sourceLinesOfFile fpIn1|]
      in2   ← source [|sourceLinesOfFile fpIn2|]
      aps   ← append in1 in2
      count ← fold   [|λc _ → c + 1|] [|0|] aps

      sink count     [|snkCount               |]
      sink aps       [|sinkFileOfLines fpOut  |])
  return count
```

Listing 5.11: Folderol implementation of append2

	Runtime (s)	Runtime (%)	Allocation (b)	Allocation (%)
Folderol	0.29s	100%	8.6e8	100%
Hand	0.29s	100%	8.6e8	100%
Conduit	0.93s	320%	3.3e9	383%
Pipes	0.94s	324%	3.4e9	395%
Streaming	0.57s	196%	2.6e9	302%

Table 5.4: Append2 benchmark results

support an infinite in-memory vector, but the underlying stream fusion (Coutts et al., 2007) could be used to process infinite streams.

5.1.5 *File operations*

For the file benchmarks, we compare against a hand-fused implementation, as well as the three previously mentioned Haskell streaming libraries: Conduit, Pipes, and Streaming. We do not compare against Vector.

The first file benchmark appends two files while counting the lines. Listing 5.11 shows the Folderol implementation. In Conduit (Listing A.10) and Pipes (Listing A.11), counting the lines is implemented as a stream transformer which counts each line before passing it along. Table 5.4 shows the runtimes for appending two text files each with half a million lines, around 6.5MB in total. The three streaming implementations cannot statically remove all of the streaming overhead.

```
part2 :: FilePath → FilePath → FilePath → IO (Int, Int)
part2 fpIn1 fpOut1 fpOut2 = do
  (c1,(c2,())) ← scalarIO $ λsnkC1 → scalarIO $ λsnkC2 →
    $$(fuse $ do
      in1       ← source    [|sourceLinesOfFile fpIn1|]
      (o1s,o2s) ← partition [|λl → length l `mod` 2 == 0|] in1

      c1        ← fold      [|λc _ → c + 1|] [|0|] o1s
      c2        ← fold      [|λc _ → c + 1|] [|0|] o2s

      sink c1               [|snkC1                |]
      sink c2               [|snkC2                |]
      sink o1s              [|sinkFileOfLines fpOut1|]
      sink o2s              [|sinkFileOfLines fpOut2|])
  return (c1, c2)
```

Listing 5.12: Folderol implementation of `part2`

		Runtime (s)	Runtime (%)	Allocation (b)	Allocation (%)
Folderol		0.30s	100%	8.6e8	100%
Hand		0.30s	100%	8.6e8	100%
Conduit	hand	0.66s	220%	2.4e9	279%
Pipes	hand	0.55s	183%	2.0e9	232%
Streaming		1.21s	403%	6.1e9	709%

Table 5.5: Part2 benchmark results

```
part2Conduit in1 out1 out2 =
  C.runConduit (sources C..| sinks)
 where
  sources = sourceFile in1

  sinks = do
   h1 ← lift $ IO.openFile out1 IO.WriteMode
   h2 ← lift $ IO.openFile out2 IO.WriteMode
   ij ← go h1 h2 0 0
   lift $ IO.hClose h1
   lift $ IO.hClose h2
   return ij


  go h1 h2 !c1 !c2 = do
   e ← C.await
   case e of
    Nothing   → return (c1,c2)
    Just v
     | prd v → do
      lift $ Char8.hPutStrLn h1 v
      go h1 h2 (c1 + 1) c2
     | otherwise → do
      lift $ Char8.hPutStrLn h2 v
      go h1 h2 c1 (c2 + 1)

  prd l = ByteString.length l `mod` 2 == 0
```

Listing 5.13: Conduit implementation of `part2`

```
part2Streaming in1 out1 out2 = do
  (i S.:> j S.:> _)  ← go
  return (i,j)
 where
  go
   = into        prd  out1
   $ into (not ∘ prd) out2
   $ S.copy
   $ sourceFile in1

  into p o i
   = sinkFile o
   $ S.store S.length
   $ S.filter p i

  prd l = ByteString.length l `mod` 2 == 0
```

Listing 5.14: Streaming implementation of `part2`

The second file benchmark takes a file and partitions it into two files: one with even-length lines, and one with odd-length lines. The output lines are also counted. Listing 5.12 shows the Folderol implementation. The Conduit implementation (Listing 5.13) is partially hand-fused, with the partition, counting and output implemented as a single recursive function that constructs a Conduit. Conduit still incurs some streaming overhead when transferring elements from the source, which reads from the file, to the recursive conduit. The Pipes implementation (Listing A.14) is similar to the Conduit implementation, and incurs a similar amount of streaming overhead. The Streaming implementation (Listing 5.14) allows streams to be shared in a fairly straightforward way and does not require hand-fusion, but is also the slowest in this benchmark, as the streaming overhead is not statically removed. Table 5.5 shows the runtimes for partitioning a file with one million lines, around 6.5MB.

5.1.6 *Partition / append*

We now look at an example that cannot be fused into a single process without introducing unbounded buffers or multiple traversals. For this example, the structure of the process network is more important than the details of the worker functions. The program in Listing 5.15 converts the input vector to a stream, and partitions the stream elements into a stream for the even-valued elements, and a stream for the odd-valued elements. The even-valued ele-

```
partitionAppendFailure :: Vector Int → IO (Vector Int)
partitionAppendFailure xs = do
  (apps,()) ← vectorSize xs $ λsnkApps →
    $$(fuse $ do
        x0           ← source    [|sourceOfVector xs|]
        (evens,odds) ← partition [|λi → even i     |] x0
        evens'       ← map       [|λi → i `div` 2  |] evens
        odds'        ← map       [|λi → i * 2      |] odds
        apps         ← append evens' odds'
        sink apps                [|snkApps         |])
  return apps
```

Listing 5.15: Partition / append fusion failure

ments are halved, while the odd-valued elements are doubled. The two result streams are then appended together and converted back to a vector.

If we try to compile `partitionAppendFailure` with Folderol, we get the compile-time warning in Listing 5.16. This warning indicates that the program could not be fused into a single loop without introducing unbounded buffering, and will instead be executed as a partially fused concurrent process network. The warning shows the dependencies between processes in the process network, with each process' input channels and output channels. The Template Haskell implementation does not have access to the variable names used by the original program, so communication channels are denoted by the placeholder names `C0` to `C5`. The input process network shows a line for each source, sink and process. The source `{sourceOfVector xs}` produces the `C0` stream, named `x0` in the original program. The `partition` process consumes the `C0` stream and produces the output channels `C1` and `C2`, named `evens` and `odds` respectively in the original program.

The partially fused process network shows that the `append` process has been fused with the two `map` processes, but could not be fused with the `partition` process. Some sort of buffering is inevitable for this process network, because `append` requires all the `evens'` first: all the even-valued elements need to be read from the input stream `x0` before the odd-valued elements can be read by `append`. To implement this in a single, sequential, traversal of the input stream, we would need to read from the input stream `x0`, and if we see an odd-valued element, we would need to hold on to it until we were sure there were no even-valued elements left.

In the `partitionAppendFailure` program, the input stream is backed by a real vector. We can, in fact, rewrite the program without introducing an unbounded buffer: we already have the buffer. To execute the program, we explicitly introduce another source to read from the vec-

```
bench/Bench/PartitionAppend/Folderol.hs:18:8: warning:
  Maximum process count exceeded: there are 2 processes after fusion.
  Inserting unbounded communication channels between remaining processes.

  Input process network (4 processes):
    ()    ->-{sourceOfVector xs}--> C0
    C0    ->-----(partition)------> C1 C2
    C1    ->--------(map)---------> C3
    C2    ->--------(map)---------> C4
    C3 C4 ->------(append)--------> C5
    C5    ->------{snkApps}-------> ()

  Partially fused process network (2 processes):
    ()    ->-{sourceOfVector xs}--> C0
    C0    ->-----(partition)------> C1 C2
    C1 C2 ->-(map / map / append)-> C5
    C5    ->------{snkApps}-------> ()
```

Listing 5.16: Partition / append fusion failure compile-time warning

tor and separate the `partition` process into two `filter` processes, as in Listing 5.17. From the perspective of the process network, the two `filter` processes apply to different input sources: even though they are backed by the same vector, there is no need to coordinate the two uses. All of `evens'` can be read by `append`, independently of `odds'`.

We can implement this program another way, if we are willing to introduce two intermediate arrays. In this version we introduce two loops. In the first loop, we partition the input into two intermediate arrays. Then in the second loop, we append the two intermediate arrays. This implementation is shown in Listing 5.18.

The two Vector implementations (Listing A.16) follow the same structure as the two sequential Folderol implementations: the two-loop version uses an intermediate array to store the filtered arrays in before appending them; the other indexes the input array twice, thus computing the predicate twice. The two-loop Vector version uses the `partition` primitive, which is itself a hand-fused implementation of two filters. The Vector `partition` cannot fuse with any consumers and always constructs manifest output vectors. The fact that `partition` constructs a manifest vector does allow an optimisation that Folderol cannot perform: if the size is known, the output can be filled at both ends, with the elements that satisfy the predicate filling the start of the array, and the elements that do not satisfy the predicate filling from the back of the array. Once the end of the input is reached, the second half, which was filled

```
partitionAppend2Source xs = do
  (apps,()) ← vectorSize xs $ λsnkApps →
    $$(fuse $ do
        x0           ← source [|sourceOfVector xs|]
        x1           ← source [|sourceOfVector xs|]
        evens        ← filter [|λi → even i     |] x0
        odds         ← filter [|λi → odd  i     |] x1
        evens'       ← map    [|λi → i `div` 2  |] evens
        odds'        ← map    [|λi → i * 2      |] odds
        apps         ← append evens' odds'
        sink apps       [|snkApps          |])
  return apps
```

Listing 5.17: Partition / append with two sources

```
partitionAppend2Loop xs = do
  (evens'V,(odds'V,())) ← vectorSize xs $ λsnkEvens' →
                          vectorSize xs $ λsnkOdds' →
    $$(fuse $ do
        x0           ← source    [|sourceOfVector xs|]
        (evens,odds) ← partition [|λi → even i     |] x0
        evens'       ← map       [|λi → i `div` 2  |] evens
        odds'        ← map       [|λi → i * 2      |] odds
        sink evens'              [|snkEvens'       |]
        sink odds'               [|snkOdds'        |])
  (apps,()) ← vectorSize xs $ λsnkApps →
    $$(fuse $ do
        evens' ← source    [|sourceOfVector evens'V |]
        odds'  ← source    [|sourceOfVector odds'V  |]
        apps   ← append evens' odds'
        sink apps          [|snkApps                |])
  return apps
```

Listing 5.18: Partition / append with two loops

		Runtime (s)	Runtime (%)	Allocation (b)	Allocation (%)
Folderol	2-source	0.42s	100%	8.0e7	100%
	2-loop	0.33s	79%	2.4e8	300%
	KPN partially fused (2 CPUs, chunk 10^3)	0.53s	126%	3.2e8	400%
	KPN unfused (4 CPUs, chunk 10^3)	0.82s	195%	5.7e8	713%
Vector	2-source	0.41s	98%	1.6e8	200%
	2-loop	0.31s	74%	1.6e8	200%

Table 5.6: PartitionAppend2 benchmark results

back-to-front, can be reversed in-place. The Vector program then splits the array in two before performing the `map` operations and appending the results back together.

Table 5.6 shows the runtimes for partitioning and appending an input vector with 10^7 elements. In the results table, the Vector version with two loops is the fastest, but the difference between it and Folderol with two loops is fairly small. The Folderol version with two sources uses the least memory, as it requires no intermediate arrays. The two concurrent versions are `partitionAppendFailure`, which was partially fused into the two processes shown in the warning in Listing 5.16, and a version with fusion disabled, which executes as the four original processes. We executed with the number of processors ranging from one to four, and the chunk size ranging from 10^0 to 10^6; the table shows the execution with the fastest runtime. Both concurrent versions incur some channel communication overhead for both runtime and allocation.

These results illustrate a trade-off between memory usage and throughput. Sometimes, we are willing to sacrifice lower throughput for lower memory usage, and this decision depends on the context. By giving a compile-time warning when fusion fails, we are able to maintain a high level of abstraction, while empowering the author to make these scheduling decisions.

5.2 RESULT SIZE

As with any practical fusion system, we must be careful that the size of the result code does not become too large when more and more processes are fused together. Figure 5.3 shows the maximum number of output states in the result when a particular number of processes are fused together in a pipelined-manner. To produce this graph, we programmatically generated

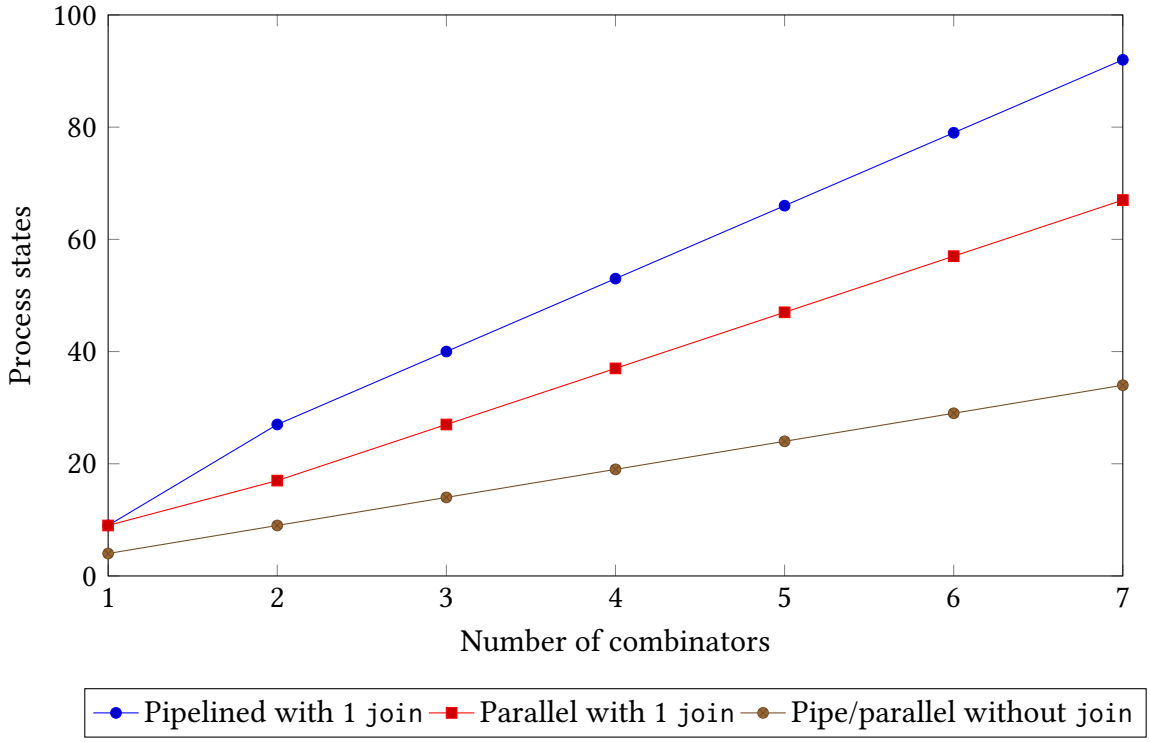


Figure 5.3: Maximum output process size for fusing all combinations of up to *n* combinators

dataflow networks for *all possible* pipelined combinations of the `map`, `filter`, `scan`, `group` and `join` combinators, and tried all possible fusion orders consiting of adjacent pairs of processes. The `join` combinator itself has two inputs, so only works at the very start of the pipeline — we present result for pipelines with and without a `join` at the start. The same graph shows the number of states in the result when the various combinations of combinators are fused in parallel, for example, we might have a `map` and a `filter` processing the same input stream. In both cases, the number of states in the result process grows linearly with the number of processes. In all combinations, with up to 7 processes there are fewer than 100 states in the result process.

The size of the result process is roughly what one would get when inlining the definitions of each of the original source processes. This is common with other systems based on inlining and/or template meta-programming, and is not prohibitive.

On the other hand, Figure 5.4 shows the results for a pathological case where the size of the output program is exponential in the number of input processes. The source dataflow networks consists of N join processes, N+1 input streams, and a single output stream. The output of each

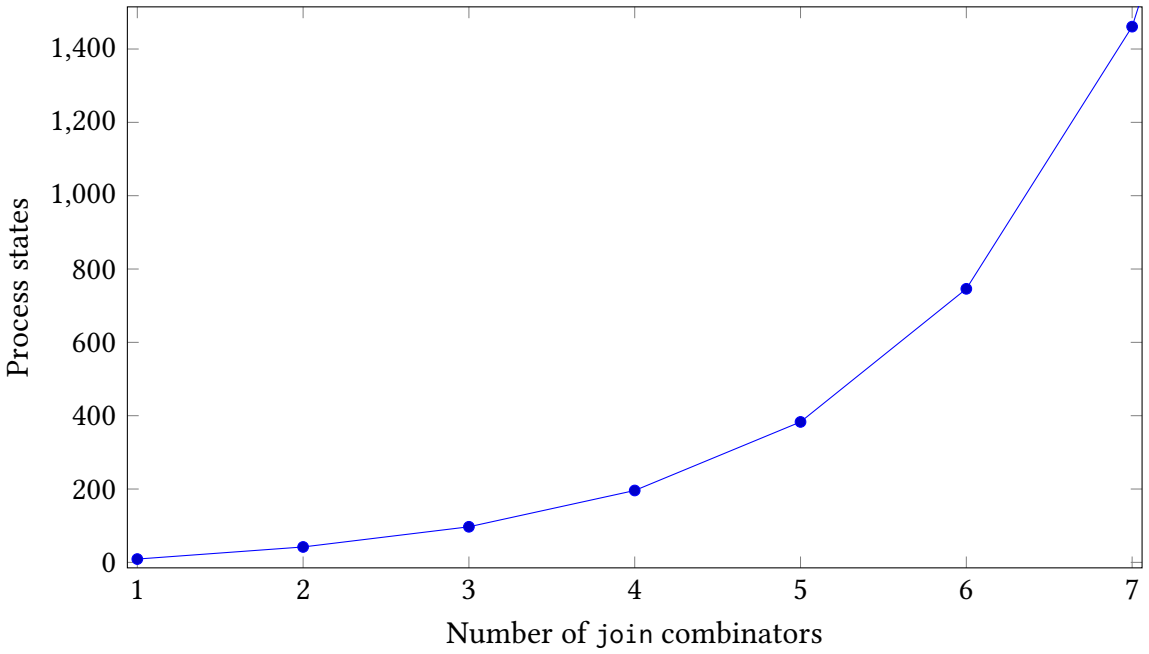


Figure 5.4: Exponential blowup occurs when splitting or chaining `join` combinators together

join process is the input of the next, forming a chain of joins. In source notation the network for N = 3 is (`sOut = join sIn1 (join sIn2 (join sIn3 sIn4))`).

When fusing two processes, the fusion algorithm essentially compares every state in the first process with every state in the second, computing a cross product. During the fusion transform, as states in the result process are generated they are added to a finite map — the `instrs` field of the process definition. The use of the finite map ensures that identical states are always combined, but genuinely different states always make it into the result.

In the worst case, fusion of two processes produces $O(n * m)$ different states, where n and m are the number of states in each. If we assume the two processes have about the same number of states then this is $O(n^2)$. Fusing the next process into this result yields $O(n^3)$, so overall the worst case number of states in the result will be $O(n^k)$, where k is the number of processes fused.

In the particular case of `join`, the implementation has two occurrences of the `push` instruction. During fusion, the states for the consuming process are inlined at each occurrence of `push`. These states are legitimately different because at each occurence of `push` the input channels of the join process are in different channel states, and these channel states are included in the overall process state. It may be possible to solve this problem by introducing the concept of subroutines, which would allow the same set of consumer instructions to be reused by the

different occurences of the push instructions. The fusion algorithm would be modified to inline the definition of a subroutine when required to coordinate between the two processes; when coordination is not required, the subroutine can be called as-is. Currently, all instructions are continuation-passing-style and the processes do not require a call-stack. To ensure subroutines can be executed with a statically bounded call-stack, subroutines must be non-recursive, but it may be possible to allow subroutines to perform loops and call other subroutines.

5.3 CONCLUSION

In the benchmarks, we saw that Folderol was always competitive with the hand-fused program, and in all but a few cases, faster than the other programs. For array computations there is some extra code required to convert between vectors and streams. There is also some extra wrapping with Template Haskell splices and quasiquotes, but this is fairly minor, and the required changes are more or less mechanical and type-driven.

Another benefit, aside from performance, is not having to inspect the compiler's intermediate representation or generated code to know whether whether fusion has worked or failed. If fusion succeeds, we know it has succeeded. If fusion fails, we are told which parts of the process network were able to be fused, and which parts were not. These explicit fusion failures make it significantly easier to track down performance issues due to non-fusion.

CHAPTER 6

RELATED WORK FOR PROCESS FUSION

This chapter discusses related work on streaming and fusion. Some of these points have been touched on previously; we now expand upon them.

6.1 FUSION AND STREAMS FOR FUNCTIONAL PROGRAMS

This thesis aims to address the limitations of combinator-based stream fusion systems to execute multiple queries concurrently. As explained in Chapter 2, neither pull-based nor push-based streams are sufficient to execute multiple queries which read from multiple inputs. To execute multiple queries, we need to be able to share streams among multiple consumers, a feature which push streams support, but pull streams do not. However, for queries containing combinators with multiple inputs such as `zip` and `join`, we can use pull streams, but not push streams.

The listlessness transform, an early form of fusion described by Wadler (1984), can execute multiple queries concurrently under some circumstances; however, the transform is not guaranteed to terminate, and is known to diverge on relatively simple programs (Caspi and Pouzet, 1996). Deforestation (Wadler, 1990), an extension of listlessness to support arbitrary recursive data types, addressed the problem of divergence by requiring the program to use each input data structure only once. This linearity constraint equates to disallowing sharing of streams.

Shortcut fusion is an attractive idea, as it allows fusion systems to be specified by a single rewrite rule. Shortcut fusion relies on local inlining, which can duplicate work when a definition is inlined into multiple use-sites. To avoid duplication of work, the Glasgow Haskell Compiler does not perform local inlining for multiple use-sites (Jones and Santos, 1998). This single use-site restriction is similar to the single consumer restriction of pull streams, and most shortcut fusion systems are pull-based. Push-based shortcut fusion systems *do* exist (Gill et al., 1993), but support neither `zip` nor `unzip` (Svenningsson, 2002; Lippmeier et al., 2013). We discussed the Vector library's use of stream fusion shortcut fusion system (Coutts et al., 2007) earlier, in Chapter 5.

Recent work on stream fusion by Kiselyov et al. (2017) uses staged computation in a pull-based system to ensure all combinators are inlined, but when streams are used multiple times this causes excessive inlining, which duplicates work. For effectful inputs such as reading from the network, duplicating work changes the semantics.

Our previous work on data flow fusion (Lippmeier et al., 2013) is neither pull-based nor push-based, and supports stream sharing and combinators with multiple inputs. It supports standard combinators such as `map`, `filter` and `fold`, and converts each stream to a series with explicit rate types, similar to the clock types of Lucid Synchrone (Benveniste et al., 2003). These rate types ensure that well-typed programs can be fused without introducing unbounded buffers. Unfusable programs trigger a compile-time error. However, data flow fusion only supports a limited set of combinators, and adding more combinators requires changing the fusion system itself.

One way to address the difference between pull and push streams is to explicitly support both separately using the polarised streams we saw in Section 2.4, as described by Bernardy and Svenningsson (2015) and Lippmeier et al. (2016). Both systems rely on stream bindings being used linearly to ensure correctness, including boundedness of buffers. These systems require manual polarity analysis of the entire dependency graph, and require complex control flow because of the switching between pulling and pushing.

Streaming IO libraries have blossomed in the Haskell ecosystem, generally based on Iteratees (Kiselyov, 2012). Libraries such as Conduit (Snoyman, 2011), Enumerator (Millikin and Vorozhtsov, 2011), Machines (Kmett et al., 2012), Pipes (Gonzalez, 2012) and Streaming (Thompson, 2015) are all designed to write stream computations with bounded buffers. These libraries all provide expressive monadic interfaces, which allow the structure of the dataflow graph to depend on the values. Because the dataflow graph can change dynamically, the streaming overhead cannot always be removed statically. Pipes performs local simplifications using rewrite rules to remove some of this overhead, while Conduit implements many of its operations in terms of stream fusion (Coutts et al., 2007). Some overhead remains, and programs tend to be written over chunks of data to offset the streaming overhead. For the most part these libraries support only straight-line computations with limited branching, while Streaming supports explicit duplication in a similar way to polarised streams. We compared the runtime performance of some of these libraries earlier, in Chapter 5. In our comparison, these libraries involved significantly more overhead than our statically fused process network implementation.

The benefits of fusion have been known about for a long time, since at least the 1970s. Jackson Structured Programming (Jackson, 1975, 2002) is a design methodology where the structure of a program is derived using the structure of the input files to process. Jackson employs a method called "program inversion" that performs a similar role to fusion, by removing intermediate results. This method is similar to converting a pull computation into a push computation. While these methods were generally performed by hand, the concept is similar to mechanised fusion.

6.2 TUPLING

Tupling combines multiple traversals over a data structure into a single traversal. Tupling is more general than stream fusion: it supports simplifying multiple traversals of trees and other data structures, rather than just streams. Two types of tupling are *fold/unfold tupling* and *hylomorphism-based tupling*.

Fold/unfold tupling, such as Chiba et al. (2010), works by repeatedly unfolding or inlining a definition into its use site, performing some local rewrite-based optimisations, then re-folding the definition. The unfolding may expose some simplification opportunities, which the local rewrite rules simplify away. However, because the definitions to be unfolded are recursive, significant effort must be taken to ensure only *finite* unfoldings are generated; for this reason, Hu et al. (1997) declare fold/unfold tupling to be impractical.

Hylomorphism-based tupling, such as Hu et al. (1996a), works by expressing traversals of the data structure as a *hylomorphism*. A hylomorphism describes how to generate some intermediate structure based on the input structure, as well as describing how to fold over the intermediate structure to compute the result. The hylomorphism allows us to compute the result without generating the intermediate structure in full. If two traversals of an input data structure can be expressed as folds over the same intermediate structure, both traversals can be computed together. Tupling algorithms attempt to automatically derive a hylomorphism for a given input data structure and traversal function, but these algorithms only work for a limited set of functions. The algorithm in Launchbury and Sheard (1995) is not total and cannot fuse a `zip` combinator with both of its consumers. The language in Hu et al. (1996b) is restricted to ensure totality of the algorithm, but cannot express the data-dependent access pattern of the `join` combinator described in Section 2.2.

6.3 NEUMANN PUSH MODEL

The push streams described in Chapter 2 are different from the push model used for database execution, as introduced in Neumann (2011). In an attempt to avoid confusion, we call this the *Neumann push model*. In the Neumann push model, a stream producer is represented as a continuation which takes a sink, or push function, to push values into:

```
data PushModel a = PushModel ((a → IO ()) → IO ())
```

The consumer provides a sink by calling the continuation, then the producer repeatedly pushes all its values to the provided sink. In this model, the consumer tells the producer when to start producing the entire stream: this is in contrast with pull streams, where the consumer asks for a single element at a time, and push streams, where the producer provides a single element at a time. Neumann (2011) originally claimed that the Neumann push model was inherently more efficient than the pull model, but this claim used a comparison between a compiled Neumann push model and an un-optimised pull model (Shaikhha et al., 2018).

The control-flow for the Neumann push model is the same as that of *push arrays*, as described in Claessen et al. (2012) and Svensson and Svenningsson (2014). Here, push arrays are used as a code generation technique, with the main advantage of generating *branchless* code to append two arrays. The branchless version of append executes as two loops, one to read from each array, rather than one loop with a conditional branch inside to choose which input array to read from.

The control-flow is also the same as push-based shortcut fusion (Gill et al., 1993), as the consumer initiates the production loop. Just as push-based shortcut fusion supports neither `zip` nor `unzip` (Svenningsson, 2002), the Neumann push model does not support combinators with multiple inputs except append; nor does it support executing multiple queries concurrently.

Biboudis (2017) describes the advantage of this model when targeting the Java just-in-time (JIT) compiler, as it allows the producer to be implemented as a simple for-loop repeatedly calling the consumer function, which makes the JIT optimiser more likely to inline the consumer.

6.4 SYNCHRONISED PRODUCT AND PROCESS CALCULI

In relation to process calculi, synchronised product has been suggested as a method for fusing Kahn process networks together (Fradet and Ha, 2004), but there is no evidence that this has

been implemented or evaluated. The synchronised product of two processes allows either process to take independent or local steps at any time, but shared actions, such as when both processes communicate on the same channel, must be taken in both processes at the same time. The synchronised product operator is much simpler than our fusion algorithm, but is also much stricter. When two processes share multiple channels, synchronised product will fail unless both processes read the channels in exactly the same order. Our system can be seen as an extension of synchronised product that allows some leeway in when processes must take shared steps: the two processes do not have to take shared steps at the same time, but if one process lags behind the other, the lagging process must catch up before the leading process gets too far ahead.

It may be possible, in future work, to simplify our fusion system by preprocessing input processes to automatically insert some leeway for shared channels, before using synchronised product for fusion. We believe that this leeway can, in fact, be inserted using synchronised product itself, but our experiments in this direction have been limited.

Like synchronised product and our process fusion, *filter fusion* (Proebsting and Watterson, 1996) also statically interleaves the code of producer and consumer processes. In filter fusion, each process can have at most a single input and a single output channel; common operators like `zip`, `unzip`, `append`, `partition` and so on are not supported. Channels cannot be shared among multiple consumers. Given an adjacent producer and consumer pair, filter fusion alternately assigns control to the code of each. When the consumer needs input, control is passed to the producer; when the producer produces its value, control is passed back to the consumer. This simple scheduling algorithm works only for straight line pipelines of processes. Both synchronised product and process fusion provide a finer-grained interleaving of code, which is necessary to support combinators with multiple input streams and multiple output streams.

6.5 SYNCHRONOUS LANGUAGES

Synchronous languages such as Lustre (Halbwachs et al., 1991), Lucy-n (Mandel et al., 2010) and SIGNAL (Le Guernic et al., 2003) all use some form of clock calculus and causality analysis to ensure that programs can be statically scheduled with bounded buffers. These languages describe *passive* processes where values are fed in to streams from outside environments, such as data coming from sensors. In this case, the passive process has no control over the rate of input coming in, and if they support multiple input streams, they must accept values from them in any order. In contrast, the processes we describe are *active* processes that have control over

the input that is coming in. Active control is necessary for combinators such as the streaming `join` operator, which must choose which input to pull from on every iteration. We discussed the relationship between synchronous languages and Icicle earlier, in Section 3.5.

6.6 SYNCHRONOUS DATAFLOW

Synchronous dataflow (not to be confused with synchronous languages above) is a dataflow graph model of computation where each dataflow actor has constant, statically known input and output rates. The main advantage of synchronous dataflow is that it is simple enough for static scheduling to be decidable, but this comes at a cost of expressivity. StreamIt (Thies et al., 2002) uses synchronous dataflow for scheduling when possible, otherwise falling back to dynamic scheduling (Soule et al., 2013). Boolean dataflow and integer dataflow (Buck and Lee, 1993; Buck, 1994) extend synchronous dataflow with boolean and integer valued control ports, and attempt to recover the structure of ifs and loops from select and switch actors. These systems allow some dynamic structures to be scheduled statically, but are very rigid and only support limited control flow structures: it is unclear how streaming `join` or append could be scheduled by this system. Finite state machine-based scenario aware dataflow (FSM-SADF) (Stuijk et al., 2011; Van Kampenhout et al., 2015) is quite expressive compared to boolean and integer dataflow, while still ensuring static scheduling. A finite state machine is constructed, where each node of the FSM denotes its own synchronous dataflow graph. The FSM transitions from one dataflow graph to another based on control outputs of the currently executing dataflow graph. For example, a filter is represented with two nodes in the FSM. The dataflow graph for the initial state executes the predicate, and the value of the predicate is used to determine which transition the FSM takes: either the predicate is false and the FSM stays where it is, or the predicate is true and moves to the next state. The dataflow graph for the next state emits the value, and moves back to the first state. FSM-SADF does appear to be able to express value-dependent operations such as `join`, but lacks the composability — and familiarity — of combinators.

6.7 SUMMARY

Table 6.1 shows a summary of the features of some of the streaming models we have discussed. We limit the comparison to the more recent and closely related work. The table displays different features supported by each streaming model. The first four features denote whether par-

	Operators				Multiple consumers	Static fusion
	Append	Zip	Join	User def.		
Kahn process networks	✓	✓	✓	✓	✓	✓[C4]
Pull streams Coutts et al. (2007) Kiselyov et al. (2017)	✓	✓	✓	✓	×	✓
Push streams	![AP]	×	×	✓	✓	✓
Polarised streams Bernardy and Svenningsson (2015)	✓	✓	✓	✓	![PM]	✓
Neumann push model Biboudis (2017)	✓	×	×	✓	×	✓
Data flow fusion Lippmeier et al. (2013)	✓	✓	×	×	✓	✓
Iteratees Kiselyov (2012)	✓	✓	✓	✓	![IM]	![IF]
Synchronous languages Mandel et al. (2010)	×	✓	×	✓	✓	✓

C4 Kahn process networks can be statically fused with process fusion (Chapter 4)

AP Push streams can be appended together using a non-deterministic merge operator, if the push order can be externally controlled (Section 2.3)

PM Polarised streams can be used to distribute elements among multiple consumers, after performing manual analysis of stream polarities (Section 2.4)

IM Iteratees-style library STREAMING (Thompson, 2015) supports multiple consumers similar to polarised streams by encoding push streams as an effect in a monad transformer stack (Chapter 5)

IF Iteratees-style libraries Conduit (Snoyman, 2011) and Pipes (Gonzalez, 2012) both use rewrite rules to reduce overhead; Conduit also uses a Stream fusion representation where possible (Chapter 5)

Table 6.1: Comparison of features supported by different streaming models

ticular operators are supported, and whether new operators can be defined without changes to the model or fusion algorithm. The second-last feature denotes whether a stream can be shared among multiple consumers. The last feature denotes whether streaming overhead can be removed statically. Where support for a feature is ambiguous or requires clarification, we denote the feature with an exclamation mark (!) and include an explanatory note.

CHAPTER 7

CLUSTERING FOR ARRAY-BACKED STREAMS

This chapter presents a clustering algorithm for scheduling array programs, where the programs may perform multiple passes over input or intermediate arrays. This work was first published as Robinson et al. (2014).

In the streaming models we have seen so far, all streams are *ephemeral*: once the elements have been read, they cannot be recovered unless they are explicitly materialized into buffers. Input streams such as those that read from a network socket are ephemeral, but arrays stored in memory or in secondary storage can be re-read any number of times. For array computations, intermediate and output arrays can be stored and re-read as well. Array computations that perform multiple passes over input or intermediate arrays can be executed as multiple streaming programs. We execute each pass as its own streaming process network, fused by the process fusion algorithm from Chapter 4, with the input streams being read from arrays, and the output streams written to arrays. For a given array computation, we perform *clustering* to determine how many passes to perform, and how to schedule the individual array operations that comprise the array computation among the different passes. There are generally many possible clusterings, and the choice of clustering can affect runtime performance. To minimise the time spent reading and re-reading the data, we would like to use a clustering with as few as possible array traversals and intermediate arrays. We use *integer linear programming* (ILP), a mathematical optimisation technique, to find the best clustering according to our cost model.

The contributions of this chapter are:

- We identify an opportunity for improvement over existing imperative clustering algorithms, which do not allow *size-changing operators* such as `filter` to be assigned to the same cluster as operations that consume the output array (Section 7.1);
- We extend the clustering algorithm of Megiddo and Sarkar (1997) and Darte and Huard (2002) with support for size-changing operators. In our system, size-changing operators can be assigned to the same cluster as operations that process their input and output arrays (Section 7.4);

- We present a simplification to constraint generation that is also applicable to some ILP formulations such as Megiddo's: constraints between two nodes need not be generated if there is a fusion-preventing path between the two (Section 7.4.5);
- Our constraint system encodes the cost model as a total ordering on the cost of clusterings, expressed using weights on the integer linear program. For example, we encode that memory traffic is more expensive than the overhead of performing a separate pass, so given a choice between the two, memory traffic will be reduced (Section 7.4.4);
- We present benchmarks of our algorithm applied to several common programming patterns. Our algorithm is complete and finds the optimal clustering for the chosen cost model, which yields good results in practice. A cost model which maps exactly to program runtime performance is infeasible in general (Section 7.5).

Our implementation is available at `https://github.com/amosr/clustering`.

7.1 CLUSTERING WITH FILTERS

To see the effect of clustering, consider the following array program:

```
normalize2 :: Array Int → (Array Int, Array Int)
normalize2 xs
 = let sum1 = fold   (+)  0   xs
       gts  = filter (>   0)  xs
       sum2 = fold   (+)  0   gts
       ys1  = map    (/ sum1) xs
       ys2  = map    (/ sum2) xs
   in (ys1, ys2)
```

The `normalize2` function computes two sums: one of all the elements of `xs`, the other of only elements greater than zero. The two maps then divide each element in the input `xs` by `sum1` and `sum2`, respectively. Since we need to fully evalute the sums before we can start to execute either map, we need at least two separate passes over the input. These folds are examples of *fusion-preventing dependencies*, as the fold operator must consume its entire input stream before it can produce its result, and this result is needed before the next stream operation can begin. A fusion-preventing dependency between two combinators means that the two combinators must be assigned to different clusters.

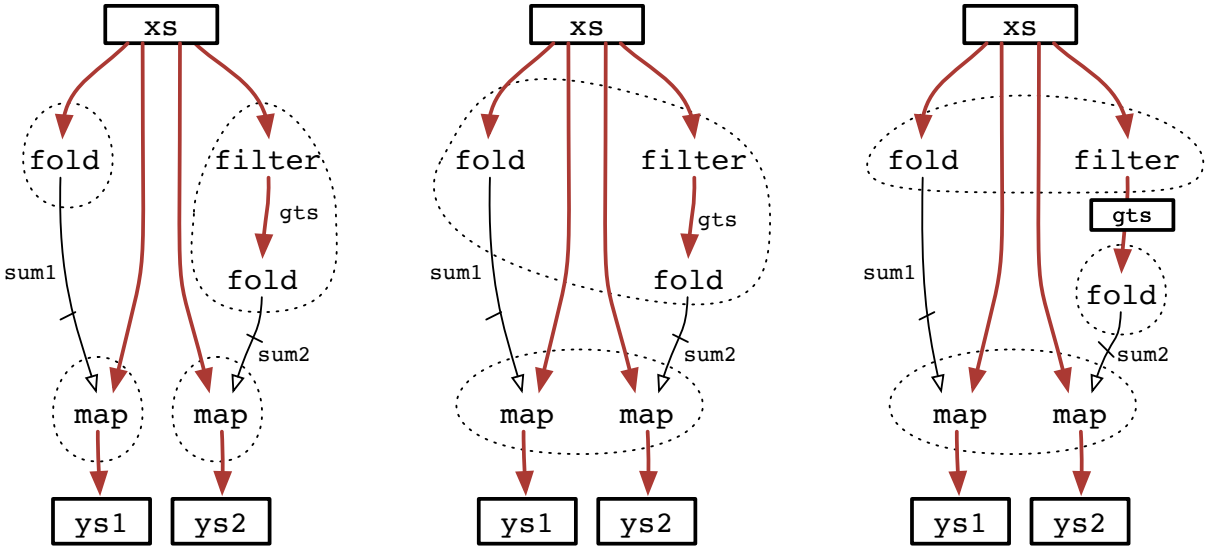


Figure 7.1: Clusterings for `normalize2`: with pull streams; our system; best imperative system

Figure 7.1 shows three cluster diagrams for `normalize2`, with each clustering produced by a different clustering algorithm. A cluster diagram is an extended version of the dependency graphs we have already seen; we explain the details in Section 7.2. The leftmost diagram shows how we have to break this program up to execute each part, assuming we use the pull stream model from Section 2.2. With pull streams we cannot compute the sums or the maps concurrently, so we end up with four loops, denoted by dotted lines in the diagram; only the filter operation is combined with the subsequent fold. If we wrote this program to use stream fusion (Coutts et al., 2007), which is a form of pull-based shortcut fusion, we would end up with the same clustering.

The rightmost diagram in Figure 7.1 shows the clustering determined by the best existing ILP approach for imperative array-based loop fusion. To obtain this clustering, we first implemented each combinator as a separate imperative loop, shown in Listing 7.1. Imperative clustering algorithms, such as Megiddo and Sarkar (1997), only cluster together loops of the same iteration size. In the imperative code, the loop that performs the fold over `gts` has an iteration size of `gts_length`, while all the other loops have an iteration size of `xs_length`. The final value of `gts_length` is not known until the loop that performs the filter completes, so there is a fusion-preventing dependency between the loop that performs the filter and the loop that performs the fold over `gts`, as well as having different iteration sizes. The low-level imperative details obscure the high-level meaning of the program, and complicate fusing the filter operation with the subsequent fold.

```
void normalize2(int* xs, size_t xs_length, int* out_ys1, int* out_ys2)
{
    // sum1 = fold (+) 0 xs
    int sum1 = 0;
    for (size_t i = 0; i != xs_length; ++i) {
        sum1 += xs[i];
    }

    // gts = filter (> 0) xs
    int* gts = malloc(sizeof(int) * xs_length);
    size_t gts_length = 0;
    for (size_t i = 0; i != xs_length; ++i) {
        if (xs[i] > 0) {
            gts[gts_length] = xs[i];
            gts_length += 1;
        }
    }

    // sum2 = fold (+) 0 gts
    int sum2 = 0;
    for (size_t i = 0; i != gts_length; ++i) {
        sum2 += gts[i];
    }
    free(gts);

    // ys1 = map (/ sum1) xs
    for (size_t i = 0; i != xs_length; ++i) {
        out_ys1[i] = xs[i] / sum1;
    }

    // ys2 = map (/ sum2) xs
    for (size_t i = 0; i != xs_length; ++i) {
        out_ys2[i] = xs[i] / sum2;
    }
}
```

Listing 7.1: Unfused imperative implementation of `normalize2`

Our approach, shown in the middle of Figure 7.1, produces the optimal clustering in this case: one loop for the filter and folds, another for the maps. For this example, we could execute each cluster as either push streams (Section 2.3) or as a process network fused by process fusion (Chapter 4). In general, a single cluster produced by our algorithm may contain combinators with multiple inputs as well as multiple outputs, so we execute each cluster as a fused process network.

7.2 COMBINATOR NORMAL FORM

To perform clustering on an input program, the program is expressed in *combinator normal form* (CNF), which is a textual description of the dependency graph. The grammar for CNF is given in Figure 7.2. Syntactically, a CNF program is a restricted Haskell function definition consisting of one or more let-bound array operations.

The `normalize2` example from Section 7.1 is already in CNF; its corresponding cluster diagrams are shown in Figure 7.1. Our cluster diagrams are similar to Loop Dependence Graphs (LDGs) from related work in imperative array fusion (Gao et al., 1993). We name edges after the corresponding variable from the CNF form, and edges which are fusion preventing are drawn with a dash through them (as per the edge labeled `sum1` in Figure 7.1). In cluster diagrams, as with dependency graphs, we tend to elide the worker functions of combinators when they are not important to the discussion — so we don't show the `(+)` operator on each use of `fold`.

Clusters of operators, which are to be fused into a single pass by process fusion, are indicated by dotted lines, and we highlight materialized arrays by drawing them in boxes. In Figure 7.1, the variables `xs`, `ys1` and `ys2` are always in boxes, as these are the material input and output arrays of the program. In the rightmost cluster diagram, `gts` has also been materialized because in this version, the producing and consuming operators (`filter` and `fold`) have not been fused together. In the grammar given in Figure 7.2, the bindings have been split into those that produce scalar values (*sbind*), and those that produce array values (*abind*). In the cluster diagrams of Figure 7.1, scalar values are represented by open arrowheads, and array values are represented by closed arrowheads.

Most of our array combinators are standard. Although not part of the grammar, we give the type of each combinator at the bottom of Figure 7.2. The $\texttt{map}_n$ combinator takes a worker function, n arrays of the same length, and applies the worker function to all elements at the same index. As such, it is similar to Haskell's `zipWith`, with an added length restriction on the argument arrays. The `generate` combinator takes an array length and a worker function, and

$$
\begin{array}{lcl}
\mathit{scalar} & \rightarrow & \text{(scalar variable)} \\
\mathit{array} & \rightarrow & \text{(array variable)} \\
f & \rightarrow & \text{(worker function)} \\
\mathit{fun} & \rightarrow & f\ \mathit{scalar}\ldots
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{bind} & ::= & \mathit{scalar} = \mathit{sbind} \\
& | & \mathit{array} = \mathit{abind} \\
& | & \mathit{scalar}\ldots, \mathit{array}\ldots = \texttt{external}\ \mathit{scalar}\ldots\ \mathit{array}\ldots
\end{array}
$$

$$
\mathit{sbind} ::= \texttt{fold}\ \ \mathit{fun}\ \ \mathit{array}
$$

$$
\begin{array}{lcl}
\mathit{abind} & ::= & \texttt{map}_n\ \ \mathit{fun}\ \ \mathit{array}^n \ \mid \texttt{filter}\ \ \mathit{fun}\ \ \mathit{array} \\
& | & \texttt{generate}\ \ \mathit{scalar}\ \ \mathit{fun} \ \mid \texttt{gather}\ \ \mathit{array}\ \ \mathit{array} \\
& | & \texttt{cross}\ \ \mathit{array}\ \ \mathit{array}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{program} & ::= & f\ \mathit{scalar}\ldots\ \mathit{array}\ldots\ = \\
& & \texttt{let}\ \mathit{bind}\ldots \\
& & \texttt{in}\ (\mathit{scalar}\ldots,\ \mathit{array}\ldots)
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{fold} & :: & (a \rightarrow a \rightarrow a) \rightarrow \texttt{Array}\ a \rightarrow a \\
\texttt{map}_n & :: & (\{a_i \rightarrow\}^{\,i \leftarrow 1\ldots n}\ b) \rightarrow \{\texttt{Array}\ a_i \rightarrow\}^{\,i \leftarrow 1\ldots n}\ \texttt{Array}\ b \\
\texttt{filter} & :: & (a \rightarrow \texttt{Bool}) \rightarrow \texttt{Array}\ a \rightarrow \texttt{Array}\ a \\
\texttt{generate} & :: & \texttt{Nat} \rightarrow (\texttt{Nat} \rightarrow a) \rightarrow \texttt{Array}\ a \\
\texttt{gather} & :: & \texttt{Array}\ a \rightarrow \texttt{Array}\ \texttt{Nat} \rightarrow \texttt{Array}\ a \\
\texttt{cross} & :: & \texttt{Array}\ a \rightarrow \texttt{Array}\ b \rightarrow \texttt{Array}\ (a, b)
\end{array}
$$

Figure 7.2: Combinator normal form

creates a new array by applying the worker to each index. The `gather` combinator takes an array of elements, an array of indices, and produces the array of elements that are located at each index. In Haskell, `gather` would be implemented as (`gather arr ixs = map (index arr) ixs`). The `cross` combinator returns the cartesian product of two arrays.

The exact form of the worker functions is left unspecified. We assume that workers are pure, can at least compute arithmetic functions of their scalar arguments, and index into arrays in the environment. We also assume that each CNF program considered for fusion is embedded in a larger host program which handles file IO and the like. Workers are additionally restricted so they can only directly reference the *scalar* variables bound by the local CNF program, though they may reference array variables bound by the host program. All access to locally bound array variables is via the formal parameters of array combinators, which ensures that all data dependencies we need to consider for fusion are explicit in the dependency graph.

The `external` binding invokes a host library function that can produce and consume arrays, but cannot be fused with other combinators. All arrays passed to and returned from host functions are fully materialised. External bindings are explicit *fusion barriers*, which force arrays and scalars to be fully computed before continuing.

Finally, note that `filter` is only one example of a size-changing operator. We can handle other size-changing operators such as `slice` in our framework, but we stick with simple filtering to aid the discussion. We discuss other combinators in Section 7.7.

7.3 SIZE INFERENCE

The array operations in a cluster are fused together into a loop with a specific number of iterations. Array operations that process different sized arrays cannot usually be assigned to the same cluster, because they require different sized loops. Consumers of arrays produced by size-changing operations can be assigned to the same cluster as operations that process different sized arrays, but only in specific circumstances, which we shall see in Section 7.3.6. Before performing clustering, we need to infer the relative sizes of each array in the program, as the sizes determine the relative loop sizes of each array operation, and whether they can be assigned to the same cluster. We use a simple constraint-based inference algorithm. Size inference has been previously described in the context of array fusion by Chatterjee et al. (1991). In constrast to our algorithm, Chatterjee et al. (1991) does not support size-changing functions such as filter.

Size Type	τ	$::=$	k	(size variable)
		$\mid$	$\tau \times \tau$	(cross product)
Size Constraint	C	$::=$	$true$	(trivially true)
		$\mid$	$k = \tau$	(equality constraint)
		$\mid$	$C \wedge C$	(conjunction)
Size Scheme	σ	$::=$	$\forall \overline{k}.\ \exists \overline{k}.\ (\overline{x : \tau}) \rightarrow (\overline{x : \tau})$	

Figure 7.3: Sizes, constraints and schemes

Although our constraint-based formulation of size inference is reminiscent of type inference for HM(X) (Odersky et al., 1999), there are important differences. Firstly, our type schemes include existential quantifiers, which express the fact that the sizes of arrays produced by filter operations are statically unknown, in general. The output size of `generate` is also statically unknown, as the result size is data-dependent and is not available until runtime. HM(X) style type inferences use the $\exists$ quantifier to bind local type variables in constraints, and existential quantifiers do not appear in type schemes. Secondly, our types are first-order only, as program graphs cannot take other program graphs as arguments. Provided we generate the constraints in the correct form, solving them is straightforward.

Size inference cannot statically infer array sizes for all programs. The $\texttt{map}_n$ combinator requires all input arrays to be the same size, and returns an output array of the same size. Compared to `zipWith`, which returns a statically unknown size, this extra restriction gives size inference more information about the size of the output array, which in turn may allow more array operations to be assigned to the same cluster. If we cannot statically determine that all input arrays given to $\texttt{map}_n$ are the same size, size inference will fail: the program may still be compiled, but fusion is not performed.

7.3.1 *Size types, constraints and schemes*

Figure 7.3 shows the grammar for size types, constraints and schemes. A size scheme is like a type scheme from Hindley-Milner style type systems, except that it only mentions the size of each input array, and ignores the element types.

A size may either be a variable k or a cross product of two sizes. We use the latter to represent the result size of the `cross` operator discussed in the previous section. Constraints

may either be trivially *true*, an equality $k = \tau$, or a conjunction of two constraints $C \wedge C$. We refer to the trivially true and equality constraints as *atomic constraints*. Size schemes relate the sizes of each input and output array. The `normalize2` example from Figure 7.1, which returns two output arrays of the same size as the input, has the following size scheme:

$$\texttt{normalize2} :_s \forall k.\, (xs : k) \to (ys_1 : k,\ ys_2 : k)$$

We write $:_s$ to distinguish size schemes from type schemes.

The existential quantifier appears in size schemes when the array produced by a filter or generate appears in the result. For example:

$\texttt{filterLeft} :_s \forall k_1.\exists k_2.(xs : k_1) \to (ys_1 : k_1, ys_2 : k_2)$

```
filterLeft xs
  = let ys1 = map (+ 1)   xs
        ys2 = filter even xs
    in (ys1, ys2)
```

The size scheme of `filterLeft` shows that it works for input arrays of all sizes. The first result array has the same size as the input, and the second has some unrelated and unknown size.

Finally, note that size schemes form only one aspect of the type information that would be expressible in a full dependently typed language. For example, in Coq or Agda we could write something like:

$\texttt{filterLeft : } \forall k_1 \texttt{ :Nat. } \exists k_2 \texttt{ :Nat. Array } k_1 \texttt{ Float} \to \texttt{(Array } k_1 \texttt{ Float, Array } k_2 \texttt{ Float)}$

However, the type inference systems for fully higher order dependently typed languages typically require quantified types to be provided by the user, and do not perform the type generalization process. In our situation, we need automatic type generalization, but for a first-order language only.

7.3.2 *Constraint generation*

The rules for constraint generation are shown in Figure 7.4. The first judgment form is written as $(\Gamma_1 \vdash \mathit{lets} \rightsquigarrow \Gamma_2 \vdash C)$ and reads: "under environment Γ_1, the bindings in *lets* produce the result environment Γ_2 and size constraints C". The judgment form $(\Gamma_1 \mid zs \vdash b \rightsquigarrow \Gamma_2 \vdash C)$ performs constraint generation for a single binding and reads: "under environment Γ_1, array

$$\boxed{\Gamma \vdash \mathit{lets} \rightsquigarrow \Gamma \vdash C}$$

$$\frac{}{\Gamma \vdash \texttt{let} \cdot \texttt{in}\ \mathit{exp} \rightsquigarrow \Gamma \vdash \mathit{true}} \quad \text{(SNil)}$$

$$\frac{\Gamma_1 \mid zs \vdash b \rightsquigarrow \Gamma_2 \vdash C_1 \quad \Gamma_2 \vdash \texttt{let}\ bs\ \texttt{in}\ \mathit{exp} \rightsquigarrow \Gamma_3 \vdash C_2}{\Gamma_1 \vdash \texttt{let}\ zs = b\ ;\ bs\ \texttt{in}\ \mathit{exp} \rightsquigarrow \Gamma_3 \vdash C_1 \wedge C_2} \quad \text{(SCons)}$$

$$\boxed{\Gamma \mid z \vdash \mathit{bind} \rightsquigarrow \Gamma \vdash C}$$

$\Gamma[xs_i : k_i]^{i \leftarrow 1..n}$	$\mid zs \vdash \texttt{map}_n\ f\ \{xs_i\}^{i \leftarrow 1..n}$	$\rightsquigarrow \Gamma,\ zs : k_{zs},\ k'$	$\vdash \bigwedge_{i \leftarrow 1..n} \{k_i = k'\} \wedge k_{zs} = k'$
Γ	$\mid zs \vdash \texttt{filter}\ f\ xs$	$\rightsquigarrow \Gamma,\ zs : k_{zs},\ \exists k'$	$\vdash k_{zs} = k'$
Γ	$\mid x \vdash \texttt{fold}\ f\ xs$	$\rightsquigarrow \Gamma$	$\vdash \mathit{true}$
Γ	$\mid zs \vdash \texttt{generate}\ s\ f$	$\rightsquigarrow \Gamma,\ zs : k_{zs},\ \exists k'$	$\vdash k_{zs} = k'$
$\Gamma[is : k_{is}]$	$\mid zs \vdash \texttt{gather}\ xs\ is$	$\rightsquigarrow \Gamma,\ zs : k_{zs},\ k'$	$\vdash k_{zs} = k',\ k_{is} = k'$
$\Gamma[xs : k_{xs},\ ys : k_{ys}]$	$\mid zs \vdash \texttt{cross}\ xs\ ys$	$\rightsquigarrow \Gamma,\ zs : k_{zs},\ k',\ k''$	$\vdash k_{zs} = k' \times k'' \wedge k_{xs} = k' \wedge k_{ys} = k''$
Γ	$\mid zs \vdash \texttt{external}\ \{xs\}^{i \leftarrow 1..n}$	$\rightsquigarrow \Gamma,\ zs : k_{zs},\ \exists k'$	$\vdash k_{zs} = k'$

Figure 7.4: Constraint generation for size inference

variable zs binds the result of b, producing a result environment Γ_2 and size constraints C". The environment (Γ) has the following grammar:

$$\Gamma ::= \cdot \mid \Gamma,\ \Gamma \mid zs : k \mid k \mid \exists k$$

As usual, $(\cdot)$ represents the empty environment and $(\Gamma,\ \Gamma)$ represents environment concatenation. The element $(zs : k)$ records the size k of some array variable zs. A plain k indicates that k can be unified with other size types when solving constraints, whereas $\exists k$ indicates a *rigid* size variable that cannot be unified with other sizes. We use the $\exists k$ syntax because this variable will also be existentially quantified if it appears in the size scheme of the overall program.

The constraints are generated in a specific form in Figure 7.4, to facilitate the constraint solving process. For each array variable in the program, we generate a new size variable, like size k_{zs} for array variable zs. These new size variables always appear on the *left* of atomic equality constraints. For each array binding, we may also introduce unification or rigid variables; these appear on the *right* of atomic equality constraints.

The final environment and constraints generated for the `normalize2` example from Section 7.1 are as follows, with the program shown on the right:

$$
\begin{array}{lll}
 & \texttt{xs} & : k_{xs}, \\
 & \texttt{gts} & : k_{gts},\ \exists k_1, \\
 & \texttt{ys1} & : k_{ys1},\ k_2, \\
 & \texttt{ys2} & : k_{ys2},\ k_3 \\
\vdash & \mathit{true} \wedge k_{gts} = k_1 & \\
\wedge & \mathit{true} \wedge k_{xs} = k_2 \wedge k_{ys1} = k_2 & \\
 & \qquad \wedge\ k_{xs} = k_3 \wedge k_{ys2} = k_3 &
\end{array}
$$

```
normalize2 xs
 = let sum1 = fold   (+)  0   xs
       gts  = filter (>   0)  xs
       sum2 = fold   (+)  0   gts
       ys1  = map    (/ sum1) xs
       ys2  = map    (/ sum2) xs
   in (ys1, ys2)
```

To compute the constraints and environment for this example, the input environment given to constraint generation records that the input array `xs` has the corresponding size type k_{xs}. This input environment is described in Section 7.3.3. For each binding, the rules in Figure 7.4 generate a constraint and add any required array and size bindings to the environment. The `sum1` binding, a `fold`, does not bind any array variables and works for any input size, so the corresponding rule leaves the environment as-is and produces a *true* constraint. For the `gts` binding, a `filter`, the size of the output array is unknown. The `filter` rule records the size of the output array by introducing a new size-type variable k_{gts}, as well as an existential variable k_1; the rule also generates the constraint ($k_{gts} = k_1$). For the `ys1` binding, a `map`, the size of the output array is the same as the input array. The `map` rule introduces a new size variable k_{ys1} to record the size of the output array, and introduces a new unification variable k_2. The rule introduces constraints requiring the new unification variable k_2 to be equal to both the input size variable k_{xs} and the output size variable k_{ys1}. In the constraints, array size variables occur on the left-hand side and unification variables occur on the right-hand side. Constraint generation for the remaining bindings proceeds similarly.

7.3.3 *Constraint solving and generalization*

Figure 7.5 shows the rule for assigning a size scheme to a program. The top-level judgment form ($\mathit{program} :_s \sigma$) assigns size scheme σ to *program*.

Rule (SProgram) assigns a size scheme to a program by first extracting size constraints, before solving them and generalizing the result. In the rule, Γ_0 is used as the input environment to constraint generation, and is constructed by generating a fresh size variable (k_i) for each input array (xs_i). The environment and constraints produced by constraint generation are named Γ_1 and C_1; these constraints are then solved using the SOLVE function, which we describe soon. The constraints, after being solved, are stored in C_2, and the environment in Γ_2. We use the solved constraints to find the size types of the input arrays ($\overline{s}$), and the size

$$\boxed{\mathit{program} :_s \sigma}$$

$$\frac{\Gamma_0 \vdash \texttt{let}\ bs\ \texttt{in}\ \{ys_j\}^{j\leftarrow 1..m} \rightsquigarrow \Gamma_1 \vdash C_1 \qquad (\Gamma_2, C_2) = \text{SOLVE}(\Gamma_1,\ C_1) \qquad \forall k \in \text{fv}(\bar{s}).\ (\exists k) \notin \Gamma_2}{f\ \{xs\}^{i\leftarrow 1..n} = \texttt{let}\ bs\ \texttt{in}\ \{ys\}^{j\leftarrow 1..m} :_s \forall \overline{k_a}.\ \exists \overline{k_e}.\ (\{xs_i : s_i\}^{i\leftarrow 1..n}) \to (\{ys_j : t_j\}^{j\leftarrow 1..m})} \quad \text{(SProgram)}$$

$$\begin{aligned}
\text{where} \quad \Gamma_0 &= \{k_i,\ xs_i : k_i\}^{i\leftarrow 1..n} \\
\\
\bar{s} &= \{s_i \mid i \in 1..n \ \wedge\ (k_i = s_i) \in C_2\} \\
\overline{k'} &= \{k'_j \mid j \in 1..m \ \wedge\ (ys_j : k'_j) \in \Gamma_2\} \\
\bar{t} &= \{t_j \mid j \in 1..m \ \wedge\ (k'_j = t_j) \in C_2\} \\
\\
\overline{k_a} &= \{k \mid k \in \Gamma_2 \ \wedge\ k \in \text{fv}(\bar{s})\} \\
\overline{k_e} &= \{k \mid \exists k \in \Gamma_2 \ \wedge\ k \in \text{fv}(\bar{t})\}
\end{aligned}$$

Figure 7.5: Constraint solving for size inference

types of the output arrays ($\bar{t}$). We perform generalization by adding universal quantifiers for the unification variables mentioned by the types of input arrays ($\overline{k_a}$), and adding existential quantifiers for the existential variables mentioned by the types of output arrays ($\overline{k_e}$). Finally, we require that the types of input arrays do not mention any existential variables; an example of this restriction is shown in Section 7.3.4.

In the rule, the solving process is indicated by SOLVE, and takes an environment and a constraint set, and produces a solved environment and constraint set. As the constraint solving process is both standard and straightforward, we only describe it informally.

During constraint generation in the previous section, we were careful to ensure that all the size variables named after program variables are on the left of atomic equality constraints, while all the unification and existential variables are on the right. To solve the constraints, we keep finding pairs of atomic equality constraints where the same variable appears on the left, unify the right of both of these constraints, and apply the resulting substitution to both the environment and original constraints. When there are no more pairs of constraints with the same variable on the left, the constraints are in solved form and we are finished.

During constraint solving, all unification variables occuring in the environment can have other sizes substituted for them. In contrast, the rigid variables marked by the ∃ symbol cannot. For example, consider the constraints for `normalize2` mentioned before:

$$\begin{aligned} & \texttt{xs} : k_{xs},\ \texttt{gts} : k_{gts},\ \exists k_1,\ \texttt{ys1} : k_{ys1},\ k_2,\ \texttt{ys2} : k_{ys2},\ k_3 \\ \vdash\ & \mathit{true} \wedge k_{gts} = k_1 \wedge \mathit{true} \\ & \quad \wedge\ k_{xs} = k_2 \wedge k_{ys1} = k_2 \\ & \quad \wedge\ k_{xs} = k_3 \wedge k_{ys2} = k_3 \end{aligned}$$

In the highlighted constraints, k_{xs} is mentioned twice on the left of an atomic equality constraint, so we can substitute k_2 for k_3. Eliminating the duplicates, as well as the trivially *true* terms then yields:

$$\begin{aligned} & \texttt{xs} : k_{xs},\ \texttt{gts} : k_{gts},\ \exists k_1,\ \texttt{ys1} : k_{ys1},\ k_2,\ \texttt{ys2} : k_{ys2},\ k_3 \\ \vdash\ & k_{gts} = k_1 \wedge k_{xs} = k_2 \wedge k_{ys1} = k_2 \wedge k_{ys2} = k_2 \end{aligned}$$

To produce the final size scheme, we look up the sizes of the input and output variables of the original program from the solved constraints and generalize appropriately. This process is determined by the top-level rule in Figure 7.5. In the case of `normalize2`, no rigid size variables appear in the result, so we can universally quantify all size variables to get the following size scheme:

$$\texttt{normalize2} :_s \forall k_2.(xs : k_2) \to (ys_1 : k_2,\ ys_2 : k_2)$$

Rule (SProgram) also characterises the programs we accept: a program is *valid* if and only if $\exists \sigma.\ \mathit{program} :_s \sigma$.

7.3.4 *Rigid sizes*

When the environment of our size constraints contains rigid variables (indicated by $\exists k$), we introduce existential quantifiers instead of universal quantifiers into the size scheme. Consider the `filterLeft` program from Section 7.3.1:

```
filterLeft xs
 = let ys1 = map (+ 1)   xs
       ys2 = filter even xs
   in (ys1, ys2)
```

The size constraints for this program, already in solved form, are as follows:

$$\begin{aligned} & xs : k_{xs},\ ys_1 : k_{ys1},\ k_1,\ ys_2 : k_{ys2},\ \exists k_2 \\ \vdash\ & k_{xs} = k_1 \ \wedge\ k_{ys_1} = k_1 \ \wedge\ k_{ys_2} = k_2 \end{aligned}$$

As variable k_2 is marked as rigid, we introduce an existential quantifier for it, producing the size scheme stated earlier:

$$\texttt{filterLeft} :_s \ \forall k_1.\, \exists k_2.\, (xs\ :\ k_1)\ \rightarrow\ (ys_1\ :\ k_1,\, ys_2\ :\ k_2)$$

Note that, although rule (SProgram) from Figure 7.5 performs a *generalization* process, there is no corresponding instantiation rule. The size inference process works on the entire graph at a time, and there is no mechanism for one operator to invoke another. To say this another way, all subgraphs are fully inlined. Recall from Section 7.2 that we assume our operator graphs are embedded in a larger host program. We use size information to guide the clustering process, and although the host program can certainly call the operator graph, static size information does not flow across this boundary.

When producing size schemes, we do not permit the arguments of an operator graph to have existentially quantified sizes. This restriction is necessary to reject programs that we cannot statically guarantee will be well-sized. For example:

```
bad1 xs
 = let flt = filter p xs
       ys  = map2   f flt xs
   in  ys
```

The above program filters its input array, and then applies `map2` to the filtered version as well as the original array. As the `map2` operator requires both of its arguments to have the same size, `bad1` would only be valid when the predicate `p` is always true. We could execute this program as a process network if we replaced the `map2` operator with `zipWith`, and explicitly read from the input array `xs` twice, as in the two-source version of `partitionAppend` from Section 5.1.6. The result size of the `zipWith` operator is the smaller of the two input operators; the extra size restriction on `map2` simplifies size inference, as we do not need to introduce the concept of the minimum of size types. The size constraints for `bad1` are as follows:

$$\begin{aligned} & xs : k_{xs},\ flt : k_{flt},\ \exists k_1,\ ys : k_{ys},\ k_2 \\ \vdash\ & k_{flt} = k_1 \ \wedge\ k_{flt} = k_2 \ \wedge\ k_{xs} = k_2 \ \wedge\ k_{ys} = k_2 \end{aligned}$$

Solving these constraints then yields:

$$\begin{aligned} & xs : k_{xs},\ flt : k_{flt},\ \exists k_1,\ ys : k_{ys},\ k_1 \\ \vdash\ & k_{flt} = k_1 \ \wedge\ k_{xs} = k_1 \ \wedge\ k_{ys} = k_1 \end{aligned}$$

In this case, rule (SProgram) does not apply, because the parameter variable xs has size k_1, but k_1 is marked as rigid in the environment (with $\exists k_1$). This function is rejected by size inference, as a caller cannot guarantee that an input array's size matches the existential size type chosen by the function.

As a final example, the following program is ill-sized because the two filter operators are not guaranteed to produce the same number of elements:

```
bad2 xs
 = let flt1 = filter p1 xs
       flt2 = filter p2 xs
       ys   = map2   f  flt1 flt2
   in  ys
```

The initial size constraints for this program are:

$$\begin{aligned} & xs : k_{xs},\ flt1 : k_{flt1},\ \exists k_1,\ flt2 : k_{flt2},\ \exists k_2,\ ys : k_{ys},\ k_3 \\ \vdash\ & k_{flt1} = k_1 \ \wedge\ k_{flt2} = k_2 \ \wedge\ k_{flt1} = k_3 \ \wedge\ k_{flt2} = k_3 \ \wedge\ k_{ys} = k_3 \end{aligned}$$

To solve these, we note that k_{flt1} is used twice on the left of an atomic equality constraint, so we substitute k_1 for k_3:

$$\begin{aligned} & xs : k_{xs},\ flt1 : k_{flt1},\ \exists k_1,\ flt2 : k_{flt2},\ \exists k_2,\ ys : k_{ys},\ k_1 \\ \vdash\ & k_{flt1} = k_1 \ \wedge\ k_{flt2} = k_2 \ \wedge\ k_{flt2} = k_1 \ \wedge\ k_{ys} = k_1 \end{aligned}$$

At this stage we are stuck, because the constraints are not yet in solved form, and we cannot simplify them further. Both k_1 and k_2 are marked as rigid, so we cannot substitute one for the other and produce a single atomic constraint for k_{flt2}. The SOLVE function fails to return a solution, and rule (SProgram) cannot apply.

7.3.5 *Iteration size*

After inferring the size of each array variable, each operator is assigned an *iteration size*, which is the number of iterations needed in the loop that evaluates the operator. For `filter` and

other size-changing operators, the iteration and result sizes are in general different. For such an operator, we say that the result size is a *descendant* of the iteration size. Conversely, the iteration size is a *parent* of the result size.

This descendant–parent size relation is transitive, so if we filter an array, then filter the resulting array, the size of the result is a descendant of the iteration size of the initial filter. This relation arises naturally from the fact that we compile individual clusters into a single process using process fusion (Chapter 4). With process fusion, such an operation would be compiled into a process containing a single loop that pulls from a stream backed by the input array — with an iteration size identical to the size of the input array, and containing two case instructions to perform the two layers of filtering.

Iteration sizes are used to decide which operators can be fused with each other. As in prior work, operators with the same iteration size can be fused. However, in prior work, operators with different iteration size cannot be assigned to the same cluster, as imperative loop fusion systems cannot generally fuse loops of different iteration sizes. In our system, we also allow operators of different iteration sizes to be fused, provided those sizes are descendants of the same parent size.

We use T to range over iteration sizes, and write $\bot$ for the case where the iteration size is unknown. The $\bot$ size is needed to handle the `external` operator, as we cannot statically infer its true iteration size, and it cannot be fused with any other operator.

Iteration Size	T	$::= \tau$		(known size)
		$\mid \quad \bot$		(unknown size)

Once the size constraints have been solved, we can use the *iter* function in Figure 7.6 to compute the iteration size of each binding. In the definition, we use the syntax $\Gamma(xs)$ to find the $(xs : k)$ element in the environment Γ and return the associated size k. Similarly, we use the syntax $C(k)$ to find the corresponding $(k = \tau)$ constraint in C and return the associated size type τ.

7.3.6 *Transducers and compatible common ancestors*

We define the concept of *transducers* as combinators having a different output size to their iteration size. As with any other combinator, a transducer may fuse with other combinators of the same iteration size, but transducers may also fuse with combinators having iteration

$$
\begin{array}{lll}
\mathit{iter}_{\Gamma,C} & : & \mathit{bind} \to T \\
\mathit{iter}_{\Gamma,C} & \mid & (z = \texttt{fold}\ f\ xs) = C(\Gamma(xs)) \\
& \mid & (ys = \texttt{map}_n\ f\ \overline{xs}) = C(\Gamma(ys)) \\
& \mid & (ys = \texttt{filter}\ f\ xs) = C(\Gamma(xs)) \\
& \mid & (ys = \texttt{generate}\ s\ f) = C(\Gamma(ys)) \\
& \mid & (ys = \texttt{gather}\ is\ xs) = C(\Gamma(is)) \\
& \mid & (ys = \texttt{cross}\ as\ bs) = C(\Gamma(as)) \times C(\Gamma(bs)) \\
& \mid & (ys = \texttt{external}\ \overline{xs}) = \bot
\end{array}
$$

Figure 7.6: Computing the iteration size of a binding

size the same as the transducer's output size. For our set of combinators, the only transducer is `filter`.

Looking back at the `normalize2` example, the iteration sizes of the filter over `xs`, which produces the array `gts`, and the fold over `xs`, which produces the scalar `sum1`, are both k_{xs}. The iteration size of the fold over `gts`, which produces the scalar `sum2`, is k_{gts}, and the filter combinator which produces `gts` is a transducer from k_{xs} to k_{gts}. Even though k_{gts} is distinct from k_{xs}, the three combinators which produce `gts`, `sum1` and `sum2` can all be fused together.

Figure 7.7 defines a function *trans*, to find the parent transducer of a combinator application. Since each name is bound to at most one combinator, we abuse terminology here slightly and write *combinator n* when refering to the combinator occuring in the binding of the name *n*. The parent transducer $\mathit{trans}(bs, n)$ of a combinator n has the same output size as n's iteration size, but the two have different iteration sizes. Although the input program's dependency graph forms a directed acyclic graph, the relationship between the combinators in the graph and each combinator's respective parent transducer, if it has one, forms a forest.

With the *trans* function, we can express the restriction on programs we view as valid for clustering as the following:

Definition: sole transducers. If a program p is *valid*, then its bindings will have at most one transducer:

$$\forall p, \sigma, n. \quad p :_s \sigma \implies |\mathit{trans}(\mathit{binds}(p), n)| \leq 1$$

Intuitively, we can see that this restriction holds for programs on which size inference succeeds. By inspecting the definition of *trans*′ and performing case analysis on the combinator binding, we see that only the $\texttt{map}_n$ clause in *trans*′ can return multiple transducers, and only the `filter` clause directly returns a transducer. Since the constraint generation for $\texttt{map}_n$ requires all inputs to have the same size, the inputs will also have the same transducer. If the

$$
\begin{array}{lll}
\mathit{trans} & : & \{\mathit{bind}\} \to \mathit{name} \to \{\mathit{name}\} \\
\mathit{trans}(bs, o) & & \\
\quad | & o = \texttt{filter}\ f\ n \quad \in bs & = \quad \mathit{trans}'(bs, n) \\
\quad | & \text{otherwise} & = \quad \mathit{trans}'(bs, o)
\end{array}
$$

$$
\begin{array}{lll}
\mathit{trans}'(bs, o) & & \\
\quad | & o = \texttt{fold}\ f\ n \quad \in bs & = \quad \emptyset \\
\quad | & o = \texttt{map}_n\ f\ ns \quad \in bs & = \quad \bigcup_{x \in ns} \mathit{trans}(bs, x) \\
\quad | & o = \texttt{filter}\ f\ n \quad \in bs & = \quad \{o\} \\
\quad | & o = \texttt{generate}\ s\ f \quad \in bs & = \quad \emptyset \\
\quad | & o = \texttt{gather}\ i\ d \quad \in bs & = \quad \mathit{trans}(bs, i) \\
\quad | & o = \texttt{cross}\ a\ b \quad \in bs & = \quad \emptyset \\
\quad | & o = \texttt{external}\ \mathit{ins} \quad \in bs & = \quad \emptyset
\end{array}
$$

Figure 7.7: Finding the parent transducers of a combinator

inputs had different transducers, then their size would be generated by different filters, and each would have its own separate existential variable as a size type, not fulfilling the same-size requirement for $\texttt{map}_n$.

We use the ancestor transducers to determine whether two combinators of different iteration sizes may be fused together. Figure 7.8 shows the `sumPartition` example, which performs two filters over the same array, and computes the sum of each filter's output array. The unfused imperative version, shown in Listing 7.2, performs each operation as a separate loop. The last two loops compute the sums. If we look at these two loops in isolation, the relationship between the two loop iteration sizes `ys_length` and `zs_length` is unknown, and it is not pos-

```
sumPartition xs
 = let ys = filter (> 0) xs
       zs = filter (< 0) xs
       y  = fold   (+) 0 ys
       z  = fold   (+) 0 zs
   in  (a,b)
```

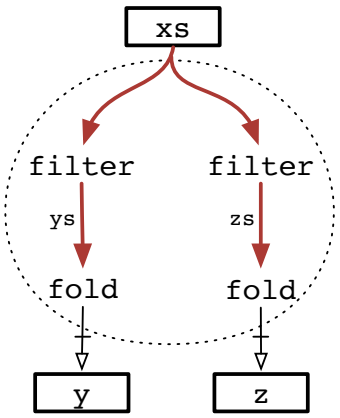


Figure 7.8: Program `sumPartition` with clustering diagram

```
void sumPartition(int* xs, size_t xs_length, int* out_y, int* out_z)
{
    // ys = filter (> 0) xs
    int* ys = malloc(sizeof(int) * xs_length);
    size_t ys_length = 0;
    for (size_t i = 0; i != xs_length; ++i) {
        if (xs[i] > 0) ys[ys_length++] = xs[i];
    }

    // zs = filter (< 0) xs
    int* zs = malloc(sizeof(int) * xs_length);
    size_t zs_length = 0;
    for (size_t i = 0; i != xs_length; ++i) {
        if (xs[i] < 0) zs[zs_length++] = xs[i];
    }

    // y = fold (+) 0 ys
    int y = 0;
    for (size_t i = 0; i != ys_length; ++i) {
        y += ys[i];
    }
    *out_y = y;

    // z = fold (+) 0 zs
    int z = 0;
    for (size_t i = 0; i != zs_length; ++i) {
        z += zs[i];
    }
    *out_z = z;

    free(ys);
    free(zs);
}
```

Listing 7.2: Unfused imperative implementation of `sumPartition` with colour-coded iteration sizes

```
void filter2(int* xs, size_t xs_length, int* out_y, int* out_z)
{
    // ys = filter (> 0) xs
    // y = fold (+) 0 ys
    int y = 0;
    for (size_t i = 0; i != xs_length; ++i) {
        if (xs[i] > 0) y += xs[i];
    }
    *out_y = y;

    // zs = filter (< 0) xs
    // z = fold (+) 0 zs
    int z = 0;
    for (size_t i = 0; i != xs_length; ++i) {
        if (xs[i] < 0) z += xs[i];
    }
    *out_z = z;
}
```

Listing 7.3: Partially fused imperative implementation of `sumPartition` with colour-coded iteration sizes

sible to fuse the two loops together without introducing excessively complicated control-flow. The relationship between the two loop iteration sizes becomes clear when we consider each sum's parent transducer, which for the `y` sum is the filter that produces `ys`, and for the `z` sum is the filter that produces `zs`. Listing 7.3 shows a partially fused imperative version, where each sum is fused with its parent transducer. In this partially fused version, the two sums are still computed in separate loops, but both loops use the same iteration size. Fusing these two loops together is trivial.

The key insight from this example is that two combinators of different iteration sizes may be fused together if each combinator is fused with its parent transducer, and the two parent transducers are also fused together. If the two parent transducers also have different iteration sizes, their respective parent transducers must also be fused together. In this case, we skip the parent transducers, and look directly at the closest pair of ancestor transducers with the same iteration size as each other.

Figure 7.9 defines the *concestors* function, which finds the pair of most recent common ancestor transducers such that both ancestors have the same iteration size. In the field of biological systematics, the term *concestor* is defined as the most recent common ancestor; here,

$$
\begin{array}{l}
concestors :: \{bind\} \to name \to name \to (name \times name)_\perp \\
concestors(bs, a, b) \\
\quad | \;\; (p_a, p_b, d) \in concestors'(bs, a, b) \\
\qquad = \{(a, b)\} \\
\quad | \;\; \text{otherwise} \\
\qquad = \perp
\end{array}
$$

$$
\begin{array}{l}
concestors' :: \{bind\} \to name \to name \to (name \times name \times \mathbb{N})_\perp \\
concestors'(bs, a, b) \\
\quad | \;\; iter_{\Gamma,C}(bs(a)) == iter_{\Gamma,C}(bs(b)) \\
\qquad = \{(a, b, 0)\} \\
\quad | \;\; a' \in trans(bs, a),\ p_a \in concestors'(bs, a', b) \\
\quad , \;\; b' \in trans(bs, b),\ p_b \in concestors'(bs, a, b') \\
\qquad = increment(closest(p_a, p_b)) \\
\quad | \;\; a' \in trans(bs, a),\ p_a \in concestors'(bs, a', b) \\
\qquad = increment(p_a) \\
\quad | \;\; b' \in trans(bs, b),\ p_b \in concestors'(bs, a, b') \\
\qquad = increment(p_b) \\
\quad | \;\; \text{otherwise} \\
\qquad = \perp
\end{array}
$$

$$
\begin{array}{l}
closest \;\; : \;\; (name \times name \times \mathbb{N}) \to (name \times name \times \mathbb{N}) \to (name \times name \times \mathbb{N}) \\
closest((l_a, l_b, l_d), (r_a, r_b, r_d)) \\
\quad | \;\; l_d \le r_d \qquad = (l_a, l_b, l_d) \\
\quad | \;\; \text{otherwise} \qquad = (r_a, r_b, r_d)
\end{array}
$$

$$
\begin{array}{l}
increment : (name \times name \times \mathbb{N}) \to (name \times name \times \mathbb{N}) \\
increment((a, b, d)) = (a, b, d + 1)
\end{array}
$$

Figure 7.9: Finding the compatible concestors, or most recent common ancestors with the same iteration size

```
normalize2 :: Array Int → (Array Int, Array Int)
normalize2 xs
 = let sum1 = fold   (+)  0   xs
       gts  = filter (>   0)  xs
       sum2 = fold   (+)  0   gts
       ys1  = map    (/ sum1) xs
       ys2  = map    (/ sum2) xs
   in (ys1, ys2)
```

Listing 7.4: Normalize2 function

we define the *compatible concestors* of a transducer to be the pair of most recent common ancestors with the same iteration size. In the definition of the *concestors* function, we use the syntax $(name \times name)_\perp$ to denote an optional pair of names. Two combinators a and b of different iteration size may be fused together only if they have compatible concestors $(c, d) \in concestors(a, b)$, and the combinators and their compatible concestors are also fused together. That is, in order for a and b to be fused together, c and d must be fused, a and c must be fused, and d and b must be fused. If the two combinators have the same iteration size, the two compatible concestors will be the combinators themselves, and the above requirements for fusing different sized combinators are trivially satisfied, since a combinator is always fused with itself.

The definition of *concestors′* returns the pair of compatible concestors as well as the distance, determined by counting how many other ancestor transducers there are between the combinator and the compatible concestors. The *closest* function compares the distances of two pairs of compatible concestors and chooses the closest pair. The *increment* function increases the distance by one.

The *trans* function returns only the direct parent transducer, but a single combinator can have multiple ancestor transducers. For a pair of combinators with multiple ancestor transducers, there may be multiple compatible common ancestors, but there can only be one pair of *most recent* compatible common ancestors (compatible concestors), because of the tree structure of ancestor transducers. In general, a pair of differently-sized combinators could be fused together if *any* pair of compatible common ancestors are fused together with the combinators. Because fusing the combinators with any pair of compatible common ancestors requires fusing with the compatible concestors as well, it is sufficient to require the pair of combinators to be fused with just the compatible concestors.

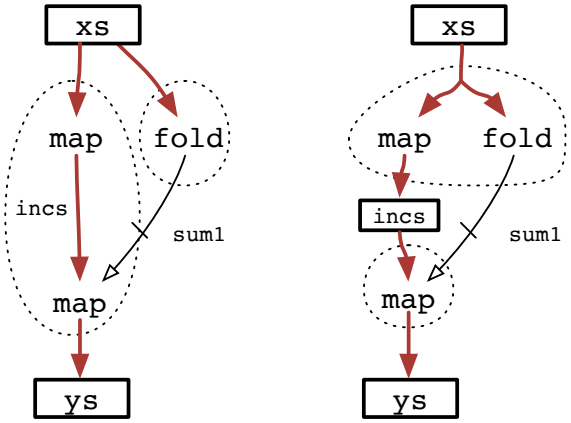


Figure 7.10: Two clusterings for `normalizeInc`

The `normalize2` example from earlier is repeated in Listing 7.4. In this example, `sum1` consumes the input `xs`, while `sum2` consumes the output of the filter `gts`, which in turn consumes the input `xs`. The two folds have different iteration sizes, and their compatible concestors are *concestors*(`sum1`, `sum2`) = (`sum1`, `gts`). The compatible concestors `sum1` and `gts` both consume the input `xs` and have the same iteration size. In order for `sum1` and `sum2` to be fused together, we require that: `sum1` and `gts` are fused together; `sum1` and `sum1` are fused together, which is trivial as fusion is reflexive; and `gts` and `sum2` are fused together.

7.4 INTEGER LINEAR PROGRAMMING

It is usually possible to cluster a program graph in multiple ways. For example, consider the following simple function:

```
normalizeInc :: Array Int → Array Int
normalizeInc xs
 = let incs = map  (+1)     us
       sum1 = fold (+) 0    us
       ys   = map  (/ sum1) incs
   in  ys
```

Two possible clusterings are shown in Figure 7.10. One option is to compute `sum1` first and fuse the computation of `incs` and `ys`. Another option is to fuse the computation of `incs` and `sum1` into a single loop, then compute `ys` separately. A third option (not shown) is to compute all results separately, and not perform any fusion.

Which option is better? On current hardware, we generally expect the cost of memory access to dominate runtime. The first clustering in Figure 7.10 requires two reads from array `xs` and one write to array `ys`. The second clustering requires a single fused read from `xs`, one write to `incs`, a read back from `incs` and a final write to `ys`. From the size constraints of the program, we know that all intermediate arrays have the same size, so we expect the first clustering will peform better as it only needs three array accesses per element in the input array, instead of four.

For small programs such as `normalizeInc`, it is possible to naively enumerate all possible clusterings, select just those that are *valid* with respect to fusion-preventing edges, and choose the one that maximises a cost metric such as the number of array accesses needed. However, as the program size increases the number of possible clusterings becomes too large to naively enumerate. For example, Pouchet et al. (2010, 2011) present a fusion system using the polyhedral model and report that some simple numeric programs have over 40,000 possible clusterings, with one particular example having 10^{12} clusterings.

To deal with the combinatorial explosion in the number of potential clusterings, we instead use an integer linear programming (ILP) formulation. ILP problems are defined as a set of variables, an objective linear function and a set of linear constraints. The integer linear solver finds an assignment to the variables that minimises the objective function, while satisfying all constraints. For the clustering problem, we express our constraints regarding fusion-preventing edges as linear constraints on the ILP variables, then use the objective function to encode our cost metric. This general approach was first fully described by Megiddo and Sarkar (1997); our main contribution is to extend their formulation to work with size-changing operators such as `filter`.

7.4.1 *Dependency graphs*

A dependency graph represents the data dependencies of the program to be fused, and we use it as an intermediate stage when producing linear constraints for the ILP problem. The dependency graph contains enough information to determine the possible clusterings of the input program, while abstracting away from the exact operators used to compute each intermediate array. The rules for producing dependency graphs are shown in Figure 7.11.

Each binding in the source program becomes a node (V) in the dependency graph. For each intermediate variable, we add a directed edge (E) from the binding that produces a value to all bindings that consume it. Each edge is also marked as either *fusible* or *fusion-preventing*.

$$
\begin{array}{lll}
V & ::= & (name \times T) \\
E & ::= & (name \times name \times E_T) \\
E_T & ::= & \text{fusible} \mid \text{fusion-preventing}
\end{array}
$$

$$
\begin{array}{lll}
nodes & : & program \to \{V\} \\
nodes(bs) & = & \{(name(b), iter_{\Gamma,C}(b)) \mid b \in bs\}
\end{array}
$$

$$
\begin{array}{lll}
edges & : & program \to \{E\} \\
edges(bs) & = & \bigcup_{b \in bs} edge(bs, b)
\end{array}
$$

$$
\begin{array}{lll}
edge & : & \{bind\} \times bind \to \{E\} \\
edge(bs, out = b) & & \\
\quad \mid \texttt{fold} \;\; f \; in & & \leftarrow b \\
\quad \mid \texttt{map} \;\; f \; in & & \leftarrow b \\
\quad \mid \texttt{filter} \;\; f \; in & & \leftarrow b \\
\quad = \{inedge(bs, out, s) \mid s \in \text{fv}(f)\} \cup \{inedge(bs, out, in)\} & & \\
\quad \mid \texttt{gather} \; data \; indices & & \leftarrow b \\
\quad = \{(out, data, \text{fusion-preventing})\} \cup \{inedge(bs, out, indices)\} & & \\
\quad \mid \texttt{cross} \; a \; b & & \leftarrow b \\
\quad = \{inedge(bs, out, a)\} \cup \{(out, b, \text{fusion-preventing})\} & & \\
\quad \mid \texttt{external} \; ins & & \leftarrow b \\
\quad = \{(o, i, \text{fusion-preventing}) \mid o \in out, \; i \in ins\} & &
\end{array}
$$

$$
\begin{array}{lll}
inedge & : & \{bind\} \times name \times name \to E \\
inedge(bs, to, from) & & \\
\quad \mid (from = \texttt{fold} \; f \; s) \in bs & & \\
\quad = (to, from, \text{fusion-preventing}) & & \\
\quad \mid (outs = \texttt{external} \ldots) \in bs \wedge from \in outs & & \\
\quad = (to, outs, \text{fusion-preventing}) & & \\
\quad \mid otherwise & & \\
\quad = (to, from, \text{fusible}) & &
\end{array}
$$

Figure 7.11: Dependency Graphs from Programs

$$\begin{array}{lcll} x & : & V \times V & \to \mathbb{B} \\ \pi & : & V & \to \mathbb{R} \\ c & : & V & \to \mathbb{B} \end{array}$$

Figure 7.12: Definition of variables in the integer linear program

Fusion-preventing edges are used when the producer must finish its execution before the consumer node can start. For example, a `fold` operation must complete execution before it can produce the scalar value needed by its consumers. Conversely, the `map` operation produces an output value for each value it consumes; as the input array is processed from start to end and values are produced incrementally, its edge is marked as fusible.

The `gather` operation is a hybrid: it takes an indices array and an elements array, and for each element in the indices array returns the corresponding data element. This means that `gather` can be fused with the operation that produces its indices, but not the operation that produces its elements — because those are accessed in a random-access manner.

The `cross` operation computes the cartesian product by looping over its second array for every element in the first array.

7.4.2 *Integer linear program variables*

After generating the dependency graph, the next step is to produce a set of linear constraints from this graph. The variables involved in these constraints are split into three groups, shown in Figure 7.12.

The first group of variables, x, is parameterised by a pair of nodes from the dependency graph. For each pair of nodes with indices i and j, we use a boolean variable $x_{i,j}$, which indicates whether those two nodes are fused. We use $x_{i,j} = 0$ when the nodes are fused and $x_{i,j} = 1$ when they are not. Using 0 for the fused case means that the objective function can be a weighted function of the $x_{i,j}$ variables, and minimizing it tends to increase the number of nodes that are fused. The values of these variables are used to construct the final clustering, such that $\forall i, j.\ x_{i,j} = 0 \iff \mathit{cluster}(i) = \mathit{cluster}(j)$.

The second group of variables in Figure 7.12, π, is parameterised by a single node index. This group of variables is used to ensure that the clustering is acyclic. An acyclic clustering is necessary to be able to execute the resulting clustering: we need to ensure that for each node in the graph, the dependencies of that node can be executed before the node itself. For each

```
mapFoldMap xs
 = let ys  = map  (+1)   xs
       sum = fold (+) 0  ys
       zs  = map  (+sum) ys
   in  zs
```

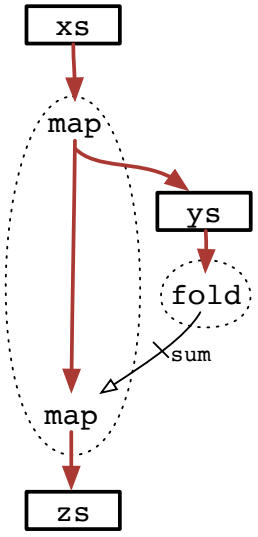


Figure 7.13: Program `mapFoldMap`, with an example of a cyclic clustering

node i, we associate a real number π_i, such that for every node j that depends on i and is not fused with i, we have $\pi_j > \pi_i$. Our linear constraints will ensure that if two nodes are fused into the same cluster, then their π values will be identical — though nodes in different clusters can also have the same π value.

The left of Figure 7.13 shows an example program, `mapFoldMap`, with an invalid clustering shown on the right. In this program, there is no fusion-preventing edge directly between the `ys` and `zs` bindings, but there is a fusion-preventing edge between `sum` and `zs`. The cluster diagram shows the `ys` and `zs` bindings in the same cluster, while `sum` is in a different cluster. This clustering contains a dependency cycle between the two clusters, and neither can be executed before the other. We constrain the π variables to reject this clustering, by requiring that $\pi_{\texttt{ys}} \leq \pi_{\texttt{sum}} < \pi_{\texttt{zs}}$. Since $\pi_{\texttt{ys}} < \pi_{\texttt{zs}}$, the two cannot be in the same cluster.

The final group of variables in Figure 7.12, c, is parameterised by a single node index. This group of variables is used to help define the cost model encoded by the objective function. Each node is assigned a variable c_i that indicates whether the array produced by the associated binding is *fully contracted*. When an array is fully contracted, it means that all consumers of that array are fused into the same cluster, so we have $c_i = 0 \iff (\forall (i', j) \in E.\ i = i' \implies x_{i,j} = 0)$. In the final program, each successive element of a fully contracted array can be stored in a scalar register, rather than requiring an array register or memory storage.

The names of the first two variable groups are standard; we propose the rather strained mnemonics $x_{i,j}$ denotes an e**x**tra loop between i and j; π_i denotes i's **p**osition in the topological

ordering; and c_i denotes that i's output array is fully **c**ontracted. These three variable groups are a mixture of the variables from previous work. Megiddo and Sarkar (1997)'s formulation uses the x and π groups, but does not include the c group; their cost model does not take into account fully contracted arrays. Darte and Huard (2002)'s formulation uses both c and π groups, where they are called k and ρ respectively, but does not include the x group, which is necessary for our formulation of size-changing operations.

7.4.3 *Linear constraints*

We place linear constraints on the integer linear program variables, and split the constraints into four groups: constraints that ensure the clustering is acyclic; constraints that encode fusion-preventing edges; constraints describing when nodes with different iteration sizes can be fused together; and constraints involving array contraction.

Acyclic and precedence-preserving constraints

The first group of constraints ensures that the clustering is acyclic, using the π variable group described earlier:

$$\begin{aligned} x_{i,j} \leq\ & \pi_j - \pi_i \leq\ N \cdot x_{i,j} \quad \text{(with an edge from } i \text{ to } j\text{)} \\ -N \cdot x_{i,j} \leq\ & \pi_j - \pi_i \leq\ N \cdot x_{i,j} \quad \text{(with no edge from } i \text{ to } j\text{)} \end{aligned}$$

As per Megiddo and Sarkar (1997), the form of these constraints is determined by whether there is a dependency between nodes i and j. The N value is set to the total number of nodes in the graph.

If there is an edge from node i to node j, we use the first constraint form shown above. If the two nodes are fused into the same cluster then we have $x_{i,j} = 0$. In this case, the constraint simplifies to $0 \leq \pi_j - \pi_i \leq 0$, which forces $\pi_i = \pi_j$. If the two nodes are in *different* clusters then the constraint instead simplifies to $1 \leq \pi_j - \pi_i \leq N$. This means that the difference between the two π variables must be at least 1, and less than or equal to N. Since there are N nodes, the maximum number of clusters, when each node is assigned to its own separate cluster, is N clusters. For this clustering the difference between any two π variables would be at most N, so the upper bound of N is large enough to be safely ignored. Ignoring the upper bound, the constraint can roughly be translated to $\pi_i < \pi_j$, which enforces the acyclicity constraint.

If instead there is no edge from node i to node j, then we use the second constraint form above. As before, if the two nodes are fused into the same cluster then we have $x_{i,j} = 0$,

which forces $\pi_i = \pi_j$. If the nodes are in different clusters then the constraint simplifies to $-N \leq \pi_j - \pi_i \leq N$, which effectively puts no constraint on the π values.

Fusion-preventing edges

As per Megiddo and Sarkar (1997), if there is a fusion-preventing edge between two nodes, we add a constraint to ensure that the nodes will be placed in different clusters.

$$x_{i,j} = 1$$
(for fusion-preventing edges from i to j)

When combined with the precedence-preserving constraints earlier, setting $x_{i,j} = 1$ also forces $\pi_i < \pi_j$.

Fusion between different iteration sizes

This group of constraints restricts which nodes can be placed in the same cluster, based on their iteration size. The group has three parts. Firstly, if either of the two nodes connected by an edge have an unknown ($\perp$) iteration size, as with `external` operators, then they cannot be fused and we set $x_{i,j} = 1$:

$$x_{i,j} = 1$$
$$\text{(if } iter_{\Gamma,C}(i) = \perp \ \vee \ iter_{\Gamma,C}(j) = \perp\text{)}$$

Secondly, if the two nodes have different iteration sizes and no common ancestors with compatible iteration sizes, then they also cannot be fused and we set $x_{i,j} = 1$:

$$x_{i,j} = 1$$
$$\text{(if } iter_{\Gamma,C}(i) \neq iter_{\Gamma,C}(j) \ \wedge \ concestors(i,j) = \perp\text{)}$$

Finally, if the two nodes have different iteration sizes but *do* have compatible common ancestors, then the two nodes can be fused together if they are fused with their respective concestors, and the concestors themselves are fused together:

$$x_{a,A} \leq x_{a,b}$$
$$x_{b,B} \leq x_{a,b}$$
$$x_{A,B} \leq x_{a,b}$$
$$\text{(if } iter_{\Gamma,C}(a) \neq iter_{\Gamma,C}(b) \ \wedge \ (A,B) \in concestors(a,b)\text{)}$$

This last part is the main difference to existing ILP solutions: we allow nodes with different iteration sizes to be fused when their common ancestor transducers are fused. The actual constraints encode a "no more fused than" relationship. For example, $x_{a,A} \leq x_{a,b}$ means that nodes a and b can be no more fused than nodes a and A.

As a simple example, consider fusing an operation on filtered data with its generating filter, as in the folds from `normalize2`:

```
sum1 = fold (+) 0  xs
gts  = filter (>0) xs
sum2 = fold (+) 0  gts
```

Here, `sum1` and `sum2` have different iteration sizes, and their compatible common ancestor transducers are computed to be *concestors*(`sum1`, `sum2`) = (`sum1`, `gts`). With the above constraints, `sum1` and `sum2` may only be fused together if three requirements are satisfied: `sum1` is fused with `sum1` (trivial), `sum2` is fused with `gts`, and `sum1` is fused with `gts`.

Array contraction

The final group of constraints gives meaning to the c variables, which represent whether an array is fully contracted:

$$x_{i,j} \quad \leq c_i$$
(for all edges from i)

An array is fully contracted when all of the consumers are fused with the node that produces it, which means that the array does not need to be fully materialized in memory. As per Darte and Huard (2002)'s work on array contraction, we define a variable c_i for each array, and the constraint above ensures that $c_i = 0$ only if $\forall (i', j) \in E.\ i = i' \implies x_{i,j} = 0$. By minimizing c_i in the objective function, we favour solutions that reduce the number of intermediate arrays.

7.4.4 *Objective function*

The objective function defines the cost model of the program, and the ILP solver will find the clustering that minimizes this function while satisfying all the constraints defined in the previous section. Our cost model has three components:

- the number of array reads and writes — an abstraction of the amount of memory bandwidth needed by the program;

$$
\begin{array}{lll}
\text{Minimise} & \Sigma_{(i,j)\in E} W_{i,j} \cdot x_{i,j} & \text{(memory traffic and loop overhead)} \\
& + \Sigma_{i \in V} N \cdot c_i & \text{(removing intermediate arrays)} \\
\text{Subject to} & \ldots \text{ constraints from Section 7.4.3 } \ldots & \\
\text{Where} & W_{i,j} = N^2 \quad | \quad (i,j) \in E & \\
& & \text{(fusing } i \text{ and } j \text{ will reduce memory traffic)} \\
& W_{i,j} = N^2 \quad | \quad \exists k.(k,i) \in E \wedge (k,j) \in E & \\
& & (i \text{ and } j \text{ share an input array}) \\
& W_{i,j} = 1 \quad | \quad \texttt{otherwise} & \\
& & \text{(the only benefit is loop overhead)} \\
& N = |V| & \\
& & \text{(the number of nodes in the graph)}
\end{array}
$$

Figure 7.14: Integer linear program with objective function

- the number of intermediate arrays — an abstraction of the amount of intermediate memory needed;

- the number of distinct clusters — an abstraction of the cost of loop management instructions, which maintain loop counters and the like.

The three components of the cost model are a heuristic abstraction of the true cost of executing the program on current hardware. They are ranked in order of importance — so we prefer to minimize the number of array reads and writes over the number of intermediate arrays, and to minimize the number of intermediate arrays over the number of clusters. However, minimizing one component does not necessarily minimize any other. For example, as the fused program executes multiple array operations at the same time, in some cases the clustering that requires the least number of array reads and writes uses more intermediate arrays than strictly necessary.

We encode the ordering of the components of the cost model as different weights in the objective function. First, note that if the program graph contains N combinators (nodes) then there are at most N opportunities for fusion, and at most N intermediate arrays. We then encode the relative cost of loop overhead as weight 1, the cost of an intermediate array as weight N, and the cost of an array read or write as weight N^2. These coefficients ensure that no amount of loop overhead reduction can outweigh the benefit of removing an intermediate array, and likewise no number of removed intermediate arrays can outweigh a reduction in the number of array reads or writes. The integer linear program including the objective function is shown in Figure 7.14.

$$
\begin{aligned}
&\mathit{possible} : \mathit{name} \times \mathit{name} \to \mathbb{B} \\
&\mathit{possible}(a, b) = \forall p \in \mathit{path}(a, b) \cup \mathit{path}(b, a).\ \text{fusion-preventing} \notin p \\
&\mathit{possible}' : \mathit{name} \times \mathit{name} \to \mathbb{B} \\
&\mathit{possible}'(a, b) = \exists A, B.\ (A, B) \in \mathit{concestors}(a, b) \wedge \mathit{possible}(a, b) \\
&\qquad\qquad \wedge\ \mathit{possible}(A, a) \wedge \mathit{possible}(B, b) \wedge \mathit{possible}(A, B)
\end{aligned}
$$

Figure 7.15: Definition of *possible* function for checking fusion-preventing paths

	Unfused		Stream		Megiddo		**Ours**	
	Time	Loops	Time	Loops	Time	Loops	Time	Loops
Normalize2	0.37s	5	0.31s	4	0.34s	3	**0.28s**	**2**
Closest points	7.34s	7	6.86s	6	6.33s	4	**6.33s**	**4**
Quadtree	0.25s	8	0.25s	8	0.11s	2	**0.11s**	**2**
Quickhull	0.43s	4	0.39s	3	0.28s	2	**0.21s**	**1**

Table 7.1: Benchmark results

7.4.5 *Fusion-preventing path optimisation*

The integer linear program defined in the previous section includes more constraints than strictly necessary to define the valid clusterings. If two nodes have a path between them that includes a fusion-preventing edge, then we know upfront that they must be placed in different clusters. Figure 7.15 defines the function $\mathit{possible}(a, b)$, which determines whether there is any possibility that the two nodes a and b can be fused. Similarly, the function $\mathit{possible}'(a, b)$ checks whether there is any possibility that the compatible common ancestors of a and b may be fused.

With *possible* and $\mathit{possible}'$ defined, we refine our formulation to only generate constraints between two nodes if there is a chance they may be fused together. The final formulation of the integer linear program is shown in Figure 7.16. This refined version generates fewer constraints, and makes the job of the ILP solver easier.

7.5 BENCHMARKS

This section discusses four representative benchmarks, and gives the full ILP program of the first benchmark. These benchmarks highlight the main differences between our fusion mechanism and related work. The runtimes of each benchmark are summarized in Table 7.1. We

$$
\begin{array}{lll}
\text{Minimise} & \Sigma_{(i,j)\in E} W_{i,j} \cdot x_{i,j} + \Sigma_{i \in V} N \cdot c_i & \\
& \text{(if } \mathit{possible}(i,j)\text{)} & \\
\text{Subject to} & -N \cdot x_{i,j} \leq \pi_j - \pi_i \leq N \cdot x_{i,j} & \\
& \text{(if } \mathit{possible}(i,j) \wedge (i,j) \notin E \wedge (j,i) \notin E\text{)} & \\
& x_{i,j} \leq \pi_j - \pi_i \leq N \cdot x_{i,j} & \\
& \text{(if } \mathit{possible}(i,j) \wedge (i,j,\text{fusible}) \in E\text{)} & \\
& \pi_i < \pi_j & \\
& \text{(if } (i,j,\text{fusion-preventing}) \in E\text{)} & \\
& x_{i,j} \leq c_i & \\
& \text{(if } (i,j,\text{fusible}) \in E\text{)} & \\
& c_i = 1 & \\
& \text{(if } (i,j,\text{fusion-preventing}) \in E\text{)} & \\
& x_{i,j} = 1 & \\
& \text{(if } \bot \in \{\mathit{iter}_{\Gamma,C}(i), \mathit{iter}_{\Gamma,C}(j)\}\text{)} & \\
& x_{i',i} \leq x_{i,j} & \\
& x_{j',j} \leq x_{i,j} & \\
& x_{i',j'} \leq x_{i,j} & \\
& \text{(if } \mathit{iter}_{\Gamma,C}(i) \neq \mathit{iter}_{\Gamma,C}(j) \wedge \mathit{possible}'(i,j) & \\
& \quad \wedge \mathit{concestors}(i,j) = \{(i',j')\}\text{)} & \\
& x_{i,j} = 1 & \\
& \text{(if } \mathit{iter}_{\Gamma,C}(i) \neq \mathit{iter}_{\Gamma,C}(j) \wedge \neg \mathit{possible}'(i,j)\text{)} & \\
\text{Where} & W_{ij} = N^2 \mid (i,j) \in E & \\
& \quad \text{(fusing } i \text{ and } j \text{ will reduce memory traffic)} & \\
& W_{ij} = N^2 \mid \exists k.(k,i) \in E \wedge (k,j) \in E & \\
& \quad (i \text{ and } j \text{ share an input array}) & \\
& W_{ij} = 1 \mid \texttt{otherwise} & \\
& \quad \text{(the only benefit is loop overhead)} & \\
& N = |V| &
\end{array}
$$

Figure 7.16: Integer linear program with fusion-preventing path optimisation

report times and the number of clusters for: the unfused case, where each operator is assigned to its own cluster; the clustering implied by pull-based stream fusion (Coutts et al., 2007); the clustering chosen by Megiddo and Sarkar (1997); and the clustering chosen by our system.

For each benchmark, we compute the different clusterings, and compile each cluster using the process fusion implementation described in Chapter 5. Using the same fusion algorithm isolates the true cost of the various clusterings from low-level differences in code generation.

We have used both GLPK (GLPK, 2013) and CPLEX (CPLEX, 2013) as external ILP solvers. For small programs such as `normalizeInc`, both solvers produce solutions in under 100ms. For a larger randomly generated example with twenty-five combinators, GLPK took over twenty minutes to produce a solution, while the commercial CPLEX solver was able to produce a solution in under one second — which is still quite usable. We will investigate the reason for this wide range in performance in future work.

The implementations of the different clusterings for the benchmark programs are available at `https://github.com/amosr/folderol/tree/bench/bench/Bench/Clustering`.

7.5.1 *Normalize2*

To demonstrate the ILP formulation, we will use the `normalize2` example from Section 7.1, repeated here:

```
normalize2 :: Array Int → (Array Int, Array Int)
normalize2 xs
 = let sum1 = fold   (+)  0   xs
       gts  = filter (>   0)  xs
       sum2 = fold   (+)  0   gts
       ys1  = map    (/ sum1) xs
       ys2  = map    (/ sum2) xs
   in (ys1, ys2)
```

We use the ILP formulation with fusion-preventing path optimisation from Section 7.4.5. First, we calculate the *possible* function to find the nodes which have no fusion-preventing path between them. The sets of nodes which can potentially be fused together are as follows:

$$\{\{sum1, gts, sum2\}, \{sum1, ys2\}, \{gts, sum2, ys1\}, \{ys1, ys2\}\}$$

$$
\begin{aligned}
\text{Minimise}\quad & 25 \cdot x_{sum1,gts} + 1 \cdot x_{sum1,sum2} + 25 \cdot x_{sum1,ys2} + \\
& 25 \cdot x_{gts,sum2} + 25 \cdot x_{gts,ys1} + 1 \cdot x_{sum2,ys1} + \\
& 25 \cdot x_{ys1,ys2} + 5 \cdot c_{gts} + 5 \cdot c_{ys1} + 5 \cdot c_{ys2}
\end{aligned}
$$

$$
\begin{aligned}
\text{Subject to}\quad & -5 \cdot x_{sum1,gts} && \leq \pi_{gts} - \pi_{sum1} && \leq 5 \cdot x_{sum1,gts} \\
& -5 \cdot x_{sum1,sum2} && \leq \pi_{sum2} - \pi_{sum1} && \leq 5 \cdot x_{sum1,sum2} \\
& -5 \cdot x_{sum1,ys2} && \leq \pi_{ys2} - \pi_{sum1} && \leq 5 \cdot x_{sum1,ys2} \\
& -5 \cdot x_{gts,ys1} && \leq \pi_{ys1} - \pi_{gts} && \leq 5 \cdot x_{gts,ys1} \\
& -5 \cdot x_{sum2,ys1} && \leq \pi_{ys1} - \pi_{sum2} && \leq 5 \cdot x_{sum2,ys1} \\
& -5 \cdot x_{ys1,ys2} && \leq \pi_{ys2} - \pi_{ys1} && \leq 5 \cdot x_{ys1,ys2} \\
& x_{gts,sum2} && \leq \pi_{sum2} - \pi_{gts} && \leq 5 \cdot x_{gts,sum2} \\
& && \pi_{sum1} < \pi_{ys1} && \\
& && \pi_{sum2} < \pi_{ys2} && \\
& x_{gts,sum2} && \leq c_{gts} && \\
& x_{gts,sum2} && \leq x_{sum1,sum2} && \\
& x_{sum1,sum1} && \leq x_{sum1,sum2} && \\
& x_{sum1,gts} && \leq x_{sum1,sum2} &&
\end{aligned}
$$

Figure 7.17: Complete integer linear program for `normalize2`

$$
\begin{aligned}
& x_{sum1,gts},\ x_{sum1,sum1},\ x_{sum1,sum2},\ x_{gts,sum2},\ x_{ys1,ys2} && = 0 \\
& x_{sum1,ys2},\ x_{gts,ys1},\ x_{sum2,ys1} && = 1 \\
& \pi_{sum1},\ \pi_{gts},\ \pi_{sum2} && = 0 \\
& \pi_{ys1},\ \pi_{ys2} && = 1 \\
& c_{gts},\ c_{ys1},\ c_{ys2} && = 0
\end{aligned}
$$

Figure 7.18: A minimal solution to the integer linear program for `normalize2`

The complete ILP program is shown in Figure 7.17. In the objective function the weights for $x_{sum1,sum2}$ and $x_{sum2,ys1}$ are both only 1, because the respective combinators do not share any input arrays.

One minimal solution to the integer linear program for `normalize2` is given in Figure 7.18. This minimal solution is not unique, though in this case the only other minimal solutions use different π values, and denote the same clustering. Looking at just the non-zero variables in the objective function, the value is $25 \cdot x_{sum1,ys2} + 25 \cdot x_{gts,ys1} + 1 \cdot x_{sum2,ys1} = 51$. For illustrative purposes, note that the objective function could be reduced by setting $x_{sum1,ys2} = 0$ (fusing $sum1$ and $ys1$), but this conflicts with the other constraints. Since $x_{sum1,sum2} = 0$, we require

that $\pi_{sum1} = \pi_{sum2}$, as well as $\pi_{sum2} < \pi_{ys2}$. These constraints cannot be satisfied, so a clustering that fused $sum1$ and $ys2$ would not also permit $sum1$ and $sum2$ to be fused.

We will now compare the clustering produced by our system with the one implied by pull-based stream fusion. As we saw in Section 2.2, pull streams do not support distributing an input stream among multiple consumers; likewise, stream fusion does not support fusing an input with multiple consumers into a single loop. The corresponding values of the x_{ij} variables are:

$$\begin{aligned} x_{gts,sum2} &= 0 \\ x_{sum1,gts}, x_{sum1,sum2}, x_{ys1,ys2}, x_{sum1,ys2}, x_{gts,ys1}, x_{sum2,ys1} &= 1 \end{aligned}$$

We can force this clustering to be applied in our integer linear program by adding the above equations as new constraints. Solving the resulting program then yields:

$$\begin{aligned} \pi_{sum1}, \pi_{gts}, \pi_{sum2} &= 0 \\ \pi_{ys1}, \pi_{ys2} &= 1 \\ c_{gts}, c_{ys1}, c_{ys2} &= 0 \end{aligned}$$

Note that although nodes $sum1$ and $sum2$ have equal π values, they are not fused because their x values are non-zero. Conversely, if two nodes have different π values, they are never fused.

For the stream fusion clustering, the corresponding value of the objective function is:
$25 \cdot x_{sum1,gts} + 1 \cdot x_{sum1,sum2} + 25 \cdot x_{sum1,ys2} + 25 \cdot x_{gts,ys1} + 1 \cdot x_{sum2,ys1} + 25 \cdot x_{ys1,ys2} = 102$.

7.5.2 *Closest points*

The closest points benchmark is a divide-and-conquer algorithm that finds the distance between the closest pair of two-dimensional points in an array. We first find the midpoint along the Y-axis, and filter the remaining points to those above and below the midpoint. We then recursively find the closest pair of points in the two halves, and merge the results.

The closest points implementation is shown in Listing 7.5, with our clustering described in the comments. To compute the clustering of this program, we ignore the base case for small arrays and only look at the recursive case. Our formulation does not directly support the `length` operator; we encode the operation to compute the `midy` midpoint as an `external` combinator when performing clustering. The two recursive calls are also encoded as `external`

```
closestPoints :: Array Point → Double
closestPoints pts
 | length pts < 100
 -- Naive O(n^2) implementation for small arrays
 = closestPointsNaive pts
 | otherwise
 = let -- (external) Midpoint
       midy    = fold (λs (x,y) → s + y) 0 pts / length pts
       -- (cluster 1) Filter above and below
       aboves  = filter (above midy) pts
       belows  = filter (below midy) pts
       -- (external) Recursive 'divide' step
       above'  = closestPoints aboves
       below'  = closestPoints belows
       border  = min above' below'
       -- (cluster 2) Find points near the border to compare against each other
       aboveB  = filter (λp → below (midy - border) p && above border p) pts
       belowB  = filter (λp → above (midy + border) p && below border p) pts
       -- (cluster 3) Merge results for 'conquer' step
       merged  = cross aboveB belowB
       dists   = map distance merged
       mins    = fold min border dists
   in  mins
```

Listing 7.5: Closest points benchmark

combinators. As the filtered points in `aboves` and `belows` are passed directly to the recursive call, there is no further opportunity to fuse them, and our clustering is the same as returned by Megiddo's algorithm. However, unlike stream fusion, our clustering fuses the filter combinators for arrays `aboves` and `belows` into a single loop, as well as fusing the filter combinators for arrays `aboveB` and `belowB`.

7.5.3 *Quadtree*

The Quadtree benchmark recursively builds a two-dimensional space partitioning tree from an array of points. We first find the initial bounding-box that all the points fit into, by computing the minimum and maximum of both X and Y axes. Then, at each recursive step, the bounding-box is split into four quadrants, and each point in the array is placed in the array for its corresponding quadrant.

The Quadtree implementation is shown in Listing 7.6. The `initialBounds` definition deconstructs the input points into two arrays containing the X and Y axes using two `map` operations. The `minimum` and `maximum` operations are implemented as folds. Our clustering for `initialBounds` fuses all six operations into a single loop, as does Megiddo's clustering. Stream fusion requires a separate loop for each fold, and would require a separate loop for each `map` operation, as each result is used twice. The Vector library, which implements stream fusion, supports constant-time `unzip` by using a struct-of-arrays representation for unboxed arrays of pairs. To provide a fair comparison, we replace the `map` operations with constant-time `unzip` in the stream fusion clustering for `initialBounds`, leading to four loops in total.

The `go` function performs the recursive loop over the points array. First, it checks whether the input array is empty or contains a single unique point, and returns a leaf node if so. Otherwise, the bounding-box is split into its four quadrants, and the points are partitioned into an array for each quadrant. Our clustering for `go` fuses all four filters into a single loop, as does Megiddo's clustering. Stream fusion requires a separate loop for each filter, with four loops in total.

7.5.4 *Quickhull*

We previously benchmarked the Quickhull algorithm in the process fusion benchmarks, in Chapter 5. Listing 7.7 shows the implementation of `filterMax`, which forms the core of the Quickhull algorithm. In our previous benchmark, we saw that stream fusion was unable to

```
quadtree :: Array Point → Quadtree
quadtree pts = go pts initialBounds
 where
  initialBounds
   = let -- (cluster 1.1) Compute initial bounds
         xs = map fst pts
         ys = map snd pts
         x1 = minimum xs
         x2 = maximum xs
         y1 = minimum ys
         y2 = maximum ys
     in (x1, y1, x2, y2)

  go ins box
   -- Non-recursive base cases for empty and small input arrays
   | length ins == 0
   = Nil
   -- Check if bounding-box contains a single point
   | smallbox box
   = LeafArray ins
   | otherwise
   = let -- (external) Split input box into four quadrants
         (b1,b2,b3,b4)    = splitbox box
         -- (cluster 2.1) Partition input points into above quadrants
         p1               = filter (inbox b1) ins
         p2               = filter (inbox b2) ins
         p3               = filter (inbox b3) ins
         p4               = filter (inbox b4) ins
         -- Recurse into each partitioned quadrant separately
     in  Tree (go p1 b1) (go p2 b2) (go p3 b3) (go p4 b4)
```

Listing 7.6: Quadtree benchmark

```
filterMax :: Line → Array Point → (Point, Array Point)
filterMax l pts
 = let -- (cluster 1) Compute filter and maximum
       ptsAnn   = map (λp → (p, distance p l)) pts
       maximAnn = maximumBy (compare `on` snd)  ptsAnn
       aboveAnn = filter ((>0) ∘ snd)           ptsAnn
       above    = map fst                       aboveAnn
   in (fst maximAnn, above)
```

Listing 7.7: Quickhull core (`filterMax`) implementation

fuse `filterMax` into a single loop because the `ptsAnn` array is used twice. Stream fusion only fuses the combinators for `aboveAnn` and `above` together, requiring three loops in total.

For `filterMax`, our clustering algorithm produces a single cluster with all combinators fused together. Megiddo's clustering for `filterMax` requires a separate loop for the `map` operation that produces the `above` array, as it is looping over the result of a filter.

7.6 RELATED WORK

The idea of using integer linear programming to cluster an operator graph for array fusion was first fully described by Megiddo and Sarkar (1997). A simpler formulation, supporting only loops of the same iteration size, but optimizing for array contraction, was then described by Darte and Huard (2002). Both algorithms were developed in the context of imperative languages (Fortran) and are based around a Loop Dependence Graph (LDG). In a LDG, the nodes represent imperative loops, and the edges indicate which loops may or may not be fused. Although this work was developed in a context of imperative programming, the conceptual framework and algorithms are language agnostic. In earlier work, Chatterjee (1993) mentioned that ILP can be used to schedule a data flow graph, though did not give a complete formulation. Our system extends the prior ILP approaches with support for size-changing operators such as `filter`.

In the loop fusion literature, the ILP approach is considered "optimal" because it can find the clustering that minimizes a global cost metric. In our case, the metric is defined by the objective function of Section 7.4.4. Besides optimal algorithms, there are also heuristic approaches. For example, Gao et al. (1993) use the maxflow-mincut algorithm to try to maximize the number of fused edges in the LDG. Kennedy (2001) describes another greedy approach

which tries to maximize the reuse of intermediate arrays, and Song et al. (2004) tries to reduce memory references.

Greedy and heuristic approaches that operate on lists of bindings rather than the graph, such as Rompf et al. (2013), can find optimal clusterings in some cases, but are subject to changes in the order of bindings. In these cases, reordering bindings can produce a different clustering, leading to unpredictable runtime performance.

Darte (1999) formalizes the algorithmic complexity of various loop fusion problems and shows that globally minimizing most useful cost metrics is NP-complete. Our ILP formulation itself is NP-hard, though in practice we have not yet found this to be a problem.

Recent literature on array fusion for imperative languages largely focuses on the polyhedral model. The polyhedral model is an algebraic representation imperative loop nests and transformations on them, including fusion transformations. Polyhedral systems (Pouchet et al., 2011) are able to express all possible distinct loop transformations where the array indices, conditionals and loop bounds are affine functions of the surrounding loop indices. However, the polyhedral model is not applicable to (or intended for) one-dimensional filter-like operations where the size of the result array depends on the source data. Recent work extends the polyhedral model to support arbitrary indexing (Venkat et al., 2014), as well as conditional control flow that is predicated on arbitrary (ie, non-affine) functions of the loop indices (Benabderrahmane et al., 2010). However, the indices used to write into the destination array must still be computed with affine functions.

Ultimately, the job of an array fusion system is to make the program go as fast as possible on the available hardware. Although the cost metrics of "optimal" fusion systems try to model the performance behavior of this hardware, it is not practical to encode the intricacies of all available hardware in a single compiler implementation. Iterative compilation approaches such as Ashby and O'Boyle (2006) instead enumerate many possible clusterings, use a cost metric to rank them, and perform benchmark runs to identify which clustering actually performs the best. An ILP formulation like ours naturally supports this model, as the integer constraints define the available clusterings, and the objective function can be used to rank them.

7.7 FUTURE WORK

One obvious opportunity for further work in our clustering formulation is to improve selection of combinators. Other combinators can currently be introduced as external computations, but

this is not ideal as they will not be fused. We now present some ideas of how to support common combinators in future work.

The size inference rules for single-input `map` can be re-used for single-input combinators that produce one output element for every input element, such as `postscanl`, `prescanl`, and `indexed`. Extraction combinators, such as `slice`, `take` and `drop`, take a contiguous 'slice' of the input. These could be implemented with an existential output size similar to the size inference for `filter`. The `tail` combinator, which discards the first element, could be implemented with an existential output size as with (`drop 1`), or perhaps by adding a new size type to denote 'decrementing' a size type.

Implementing `append` would likely involve adding a new size type for appending two size types together, similar to the result size of the `cross` combinator. Although the result size of appending two concrete input arrays is commutative, the loops that generate the two halves of the output cannot generally be interchanged. The size type for appending two inputs therefore should not be commutative. As we saw in Section 4.5.1, process fusion can fuse appends with one shared input and two different inputs into a single process. To support this clustering, the definition of transducers would have to be modified to allow fusion between nodes with the same size type as one of the inputs and nodes with the same size as the output. It may be simpler to implement `append` by introducing an existential size type for both the iteration size and the output size; however, introducing an existential for the output size hides the fact that appending two size types is an injection.

The `length` combinator is unique, as it does not require the array to be manifest, but does require some array with the same rate to be manifest. For example, finding the length of the output of a filter can only be done after the filter is computed, while finding the length of the output of a map can often be done before the map is computed. Once `length` is implemented, functions such as `reverse` can be implemented as a `generate` followed by a `gather`.

CHAPTER 8

CONCLUSION

This final chapter discusses some directions for future work before concluding the thesis. This thesis has presented five different ways to execute multiple queries concurrently, each with a different set of trade-offs.

In Section 2.3 we saw how *push streams* can be used to execute multiple queries, so long as the queries all operate over the same input. These queries were written back-to-front and the input stream must be manually duplicated whenever the input was used multiple times.

In Chapter 3 we introduced *Icicle*, an alternate presentation of push streams, using modal types to ensure that all queries over a single shared input table can be fused together and executed in a single pass. Here, values from the input stream can be shared among multiple consumers without explicitly duplicating the stream.

In Section 2.4 we saw *polarised streams*, which are a careful combination of push streams and pull streams, that can be used execute multiple queries concurrently. Writing a group of queries as polarised streams requires putting all the queries together in a single dataflow graph, then performing a manual polarity analysis on the graph, so that push or pull polarities can be assigned to each edge of the dataflow graph.

In Chapter 4 we introduced *process fusion*, where a Kahn process network is used to execute multiple queries; the processes are then fused together so the queries can be executed as a single process, without communication overhead. Here, the queries require no polarity analysis, and fusion is performed automatically. The advantage of process fusion over Icicle is that it supports multiple input streams. However, by supporting multiple input streams, we lose the guarantee that all queries over the same input can be fused together.

In Section 4.6 we gave an overview of the *mechanised proof of soundness of fusion*, which gives us confidence in the correctness of the process fusion transform.

In Chapter 5 we evaluated the runtime performance of process fusion and saw that the fused program was always at least two times faster than the compared streaming implementations, and usually between one and a half to two times faster than the compared array fusion implementation.

In Chapter 6 we compared process fusion with related work on fusion, streams and dataflow languages.

Finally, in Chapter 7 we introduced *clustering for array programs*, a scheduling algorithm for executing array programs in as few passes as possible. When the input data to the queries fits in memory as an array, we can perform multiple passes over the input data. Our clustering algorithm finds a schedule that minimises the number of array reads and writes, the number of intermediate arrays, and the number of loops. It improves on prior work by supporting size-changing operations, such as `filter`, allowing them to be fused with their consumers as well as their producers.

These methods are applicable in different situations, depending on the storage requirements of the dataset, and the kinds of queries to be executed. However, all methods are designed to reduce program runtime by minimising streaming and iteration overhead. If we wish to reduce runtime as much as possible, we cannot consider each query in isolation. We must instead consider the queries as a whole, and perform global optimisations: Icicle's intermediate representation enables common subexpression elimination across queries, process fusion coordinates a producer with all consumers, and clustering converts the entire dependency graph to an integer linear program. In contrast, shortcut fusion techniques such as stream fusion (Coutts et al., 2007) rely on local rewrites, and cannot take into account the surrounding context of the program. If we want to fuse a whole set of queries, we usually need to look at the whole set of queries, and transform them as a graph, rather than performing local transformations on the abstract syntax tree.

8.1 FUTURE WORK

We now take a brief look at the limitations and possible extensions of our work. We focus on more conceptual extensions to process fusion here; extensions to clustering to support more combinators are mentioned in Section 7.7.

8.1.1 *Network fusion order and non-determinism*

In the discussion of process fusion in Section 4.5.1, we saw that the order in which we fuse the processes in a network can affect whether fusion succeeds or fails. We propose to solve this in the future by modifying the fusion algorithm to be commutative and associative. These

properties would allow us to apply fusion in any order, knowing that all orders produce the same result.

The fusion algorithm is not commutative because when two processes are trying to execute instructions which could occur in either order, the algorithm must choose only one instruction. The fusion algorithm applies some heuristics to decide which instruction to execute first, but when evaluating the processes as a process network, the choice is non-deterministic. Fusion commits too early to a particular interleaving of the instructions from each process, when there are many possible interleavings that would work. By explicitly introducing non-determinism in the fused process, we could represent all possible interleavings, and do not have to commit to one too early. We want to stop the fusion algorithm from committing to a scheduling decision too early, and allow the result process to represent all possible schedules.

Reifying the non-determinism in the processes will mean that all fusion orders produce the same process at the end. Fusing the whole network in different orders will not affect the result, or whether processes can be fused together. The order in which the whole network is fused does affect the intermediate process, though, and some fusion orders may produce larger intermediate processes. Two unconnected processes, which read from different streams, can execute without coordinating with each other. If we fuse these two unconnected processes together, at every step the fused result process can non-deterministically choose which source process to execute. In this case, the number of distinct states for the result process is the cross product of the states in each source process. Fusing connected processes, for example a producer and a consumer, introduce less non-determinism because there are times when only one of the processes can run. When the consumer is waiting for a value, only the producer can run. With less non-determinism, the result process is likely to be smaller. We suspect that, in general, fusing connected processes will produce a smaller process than fusing unconnected processes. The size of the overall result for the entire network is the same, but the intermediate process will be smaller. Larger intermediate programs generally take longer to compile, so some heuristic order which fuses connected processes is likely to be useful, even if the order does not affect the result.

The advantage of Kahn process networks is that they guarantee a deterministic result, despite the non-deterministic evaluation order. To retain deterministic results, non-determinism must be restricted to only occur in the processes generated by the fusion algorithm, and not in the input processes defined by the user. Because the fusion algorithm is essentially a static application of the runtime evaluation rules, any non-determinism introduced by the fusion algorithm will still compute a deterministic result.

8.1.2 *Conditional branching and fusion*

In the process fusion algorithm, `case` instructions, which perform conditional branching, are simply copied to the result process. However, if both input processes branch on the same condition, we should be able to statically infer that, for example, if the first process takes the true branch, the second process should also take the true branch.

Consider the following list program, which filters the input list twice, with both filters using the same predicate:

```
filter2 :: [Int] → [(Int,Int)]
filter2 input =
 let xs  = filter (λi → i > 5) input
     ys  = filter (λi → i > 5) input
     xys = zip xs ys
 in  xys
```

If we interpret this program as a process network and try to perform fusion, the fusion algorithm fails, erroneously suggesting that the process network requires an unbounded buffer or multiple passes over the input list. This failure would be legitimate if the two filter predicates were different. However, when the predicates are the same, it *is* possible to execute with a single loop and no buffers: if the input value used by `xs` is greater than five, then the input value used by `ys` is also greater than five.

Rather than extend the fusion algorithm to track which case conditions are true at each given label, we propose to implement a separate post-processing pass to perform branch simplification on the fused process. In the `filter2` example, the fusion algorithm returns a fusion failure instead of a fused process. Implementing this extension as a separate pass would require modifying the fusion algorithm to still return the complete fused process when it detects a potential deadlock; potential deadlocks could be recorded in the result process with a new instruction. After fusing all the processes together, simplifying the result process, and removing unreachable instructions, we would check whether any deadlock instructions are reachable; if so, we trigger a fusion failure as before.

8.2 CONCLUSION

If, in the future of computing, interest in *big data* does not wane, and datasets continue to grow faster than hard drives or memory, then the concurrent execution of streaming queries will

only become more important. Efficient execution of concurrent queries has the obvious benefit of providing more immediate answers, while also improving energy efficiency. In Australia, where a large majority of our power is supplied by burning coal, we have a moral obligation to use power responsibly, as well as to use power from renewable sources where possible. Hopefully, the techniques presented in this thesis can help in a small way to reduce the overall energy consumption for querying large datasets.

BIBLIOGRAPHY

Allen, G. E., Zucknick, P. E., and Evans, B. L. (2007). A distributed deadlock detection and resolution algorithm for process networks. In *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, volume 2, pages II–33. IEEE.

Andrade, H., Aryangat, S., Kurc, T., Saltz, J., and Sussman, A. (2003). Efficient execution of multi-query data analysis batches using compiler optimization strategies. In *Languages and Compilers for Parallel Computing*.

Arasu, A., Babu, S., and Widom, J. (2002). An abstract semantics and concrete language for continuous queries over streams and relations. Technical report, Stanford InfoLab.

Arasu, A., Babu, S., and Widom, J. (2003). CQL: A language for continuous queries over streams and relations. In *Database Programming Languages*.

Ashby, T. J. and O'Boyle, M. F. P. (2006). Iterative collective loop fusion. In *CC: Compiler Construction*.

Benabderrahmane, M.-W., Pouchet, L.-N., Cohen, A., and Bastoul, C. (2010). The polyhedral model is more widely applicable than you think. In *CC: Compiler Construction*.

Benveniste, A., Caspi, P., Edwards, S. A., Halbwachs, N., Le Guernic, P., and De Simone, R. (2003). The synchronous languages 12 years later. *Proceedings of the IEEE*.

Bernardy, J.-P. and Svenningsson, J. (2015). On the duality of streams. How can linear types help to solve the lazy IO problem? In *IFL: Implementation and Application of Functional Languages*.

Biboudis, A. C. (2017). *Expressive and Efficient Streaming Libraries*. PhD thesis, Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών (ΕΚΠΑ). Σχολή Θετικών Επιστημών. Τμήμα Πληροφορικής και Τηλεπικοινωνιών. Τομέας Υπολογιστικών Συστημάτων και Εφαρμογών.

Buck, J. T. (1994). Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *Signals, Systems and Computers, Twenty-Eighth Asilomar Conference on*.

Buck, J. T. and Lee, E. A. (1993). Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on*.

Caspi, P. and Pouzet, M. (1996). Synchronous Kahn networks. In *ACM SIGPLAN Notices*, volume 31, pages 226–238. ACM.

Chatterjee, S. (1993). Compiling nested data-parallel programs for shared-memory multiprocessors. *TOPLAS: Transactions on Programming Languages and Systems*, 15(3).

Chatterjee, S., Blelloch, G. E., and Fisher, A. L. (1991). Size and access inference for data-parallel programs. In *PLDI: Programming Language Design and Implementation*.

Chen, D.-K., Su, H.-M., and Yew, P.-C. (1990). *The impact of synchronization and granularity on parallel systems*, volume 18. ACM.

Chiba, Y., Aoto, T., and Toyama, Y. (2010). Program transformation templates for tupling based on term rewriting. *IEICE TRANSACTIONS on Information and Systems*, 93(5):963–973.

Chitil, O. (1997a). Common subexpression elimination in a lazy functional language. In *Proceedings of the 9th International Workshop on Implementation of Functional Languages, St. Andrews, Scotland, September*, volume 10, page 12. Citeseer.

Chitil, O. (1997b). Common subexpressions are uncommon in lazy functional languages. In *Symposium on Implementation and Application of Functional Languages*, pages 53–71. Springer.

Chiusano, P. (2013). The fs2 Scala package. `http://fs2.io/`.

Claessen, K., Sheeran, M., and Svensson, J. (2012). Expressive array constructs in an embedded GPU kernel programming language. In *DAMP: Declarative Aspects of Multicore Programming*.

Coutts, D., Leshchinskiy, R., and Stewart, D. (2007). Stream fusion: From lists to streams to nothing at all. In *ACM SIGPLAN Notices*.

CPLEX (2013). IBM ILOG CPLEX optimizer. https://www.ibm.com/analytics/cplex-optimizer.

Darte, A. (1999). On the complexity of loop fusion. In *PACT: Parallel Architectures and Compilation Techniques*.

Darte, A. and Huard, G. (2002). New results on array contraction. In *ASAP*.

Davies, R. and Pfenning, F. (2001). A modal analysis of staged computation. *Journal of the ACM*.

Fradet, P. and Ha, S. H. T. (2004). Network fusion. In *Asian Symposium on Programming Languages and Systems*.

Gao, G., Olsen, R., Sarkar, V., and Thekkath, R. (1993). Collective loop fusion for array contraction. *Languages and Compilers for Parallel Computing*.

Geilen, M. and Basten, T. (2003). Requirements on the execution of kahn process networks. In *Programming languages and systems*.

Gill, A., Launchbury, J., and Peyton Jones, S. L. (1993). A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93. ACM.

GLPK (2013). GLPK (GNU linear programming kit). http://www.gnu.org/software/glpk.

Gonzalez, G. (2012). The pipes Haskell package. http://hackage.haskell.org/package/pipes.

Graefe, G. (1989). *Volcano: An Extensible and Parallel Query Evaluation System*. Oregon Graduate Center.

Group, S. et al. (2003). Stream: The Stanford stream data manager. *IEEE Data Engineering Bulletin*.

Gulwani, S. and Necula, G. C. (2004). A polynomial-time algorithm for global value numbering. In *Static Analysis*.

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991). The synchronous data flow programming language Lustre. *Proceedings of the IEEE*.

Hu, Z., Iwasaki, H., and Takeichi, M. (1996a). Cheap tupling transformation. *METR 96*, 8.

Hu, Z., Iwasaki, H., and Takeichi, M. (1996b). *Deriving structural hylomorphisms from recursive definitions*, volume 31. ACM.

Hu, Z., Iwasaki, H., Takeichi, M., and Takano, A. (1997). Tupling calculation eliminates multiple data traversals. *ACM Sigplan Notices*, 32(8):164–175.

Hu, Z., Yokoyama, T., and Takeichi, M. (2005). Program optimizations and transformations in calculation form. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 144–168. Springer.

Jackson, M. (2002). JSP in perspective. In *Software pioneers*, pages 480–493. Springer.

Jackson, M. A. (1975). *Principles of program design*, volume 197. Academic press London.

Jiang, B., Deprettere, E., and Kienhuis, B. (2008). Hierarchical run time deadlock detection in process networks. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 239–244. IEEE.

Johnston, W. M., Hanna, J., and Millar, R. J. (2004). Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*.

Jones, S. L. P. (1996). Compiling Haskell by program transformation: A report from the trenches. In *European Symposium on Programming*, pages 18–44. Springer.

Jones, S. L. P. and Santos, A. M. (1998). A transformation-based optimiser for Haskell. *Science of computer programming*, 32(1-3):3–47.

Kahn, G., MacQueen, D., et al. (1976). Coroutines and networks of parallel processes.

Kay, M. (2009). You pull, I'll push: on the polarity of pipelines. In *Balisage: The Markup Conference*.

Kennedy, K. (2001). Fast greedy weighted fusion. *International Journal of Parallel Programming*, 29(5).

Kiselyov, O. (2012). Iteratees. In *International Symposium on Functional and Logic Programming*.

Kiselyov, O., Biboudis, A., Palladinos, N., and Smaragdakis, Y. (2017). Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*.

Kmett, E., Bjarnason, R., and Cough, J. (2012). The machines Haskell package. `http://hackage.haskell.org/package/machines`.

Launchbury, J. and Sheard, T. (1995). Warm fusion: deriving build-catas from recursive definitions. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 314–323. ACM.

Le Guernic, P., Talpin, J.-P., and Le Lann, J.-C. (2003). Polychrony for system design. *Journal of Circuits, Systems, and Computers*.

Leshchinskiy, R. (2008). The vector Haskell package. `http://hackage.haskell.org/package/vector`.

Lippmeier, B., Chakravarty, M. M. T., Keller, G., and Robinson, A. (2013). Data flow fusion with series expressions in Haskell. In *Proceedings of the 2013 Haskell symposium*. In Submission.

Lippmeier, B., Mackay, F., and Robinson, A. (2016). Polarized data parallel data flow. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing*.

Madden, S., Shah, M., Hellerstein, J. M., and Raman, V. (2002). Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*.

Mandel, L., Plateau, F., and Pouzet, M. (2010). Lucy-n: a n-synchronous extension of Lustre. In *Mathematics of Program Construction*.

Maurer, L., Downen, P., Ariola, Z. M., and Peyton Jones, S. (2017). Compiling without continuations. In *ACM SIGPLAN Notices*, volume 52, pages 482–494. ACM.

McBride, C. and Paterson, R. (2008). Applicative programming with effects. *Journal of functional programming*, 18(1):1–13.

McSherry, F., Isard, M., and Murray, D. G. (2015). Scalability! But at what COST? In *Hot Topics in Operating Systems*.

Megiddo, N. and Sarkar, V. (1997). Optimal weighted loop fusion for parallel programs. In *SPAA: Symposium on Parallel Algorithms and Architectures*.

Millikin, J. and Vorozhtsov, M. (2011). The enumerator Haskell package. `http://hackage.haskell.org/package/enumerator`.

Munagala, K., Srivastava, U., and Widom, J. (2007). Optimization of continuous queries with shared expensive filters. In *Principles of database systems*.

Neumann, T. (2011). Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550.

Odersky, M., Sulzmann, M., and Wehr, M. (1999). Type inference with constrained types. *TAPOS*.

Parks, T. M. (1995). *Bounded scheduling of process networks*. PhD thesis, University of California. Berkeley, California.

Peyton Jones, S. (2007). Call-pattern specialisation for Haskell programs. In *ACM SIGPLAN Notices*. ACM.

Pouchet, L.-N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., and Sadayappan, P. (2010). Combined iterative and model-driven optimization in an automatic parallelization framework. In *SC: High Performance Computing, Networking, Storage and Analysis*.

Pouchet, L.-N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P., and Vasilache, N. (2011). Loop transformations: convexity, pruning and optimization. In *POPL: Principles of Programming Languages*.

Proebsting, T. A. and Watterson, S. A. (1996). Filter fusion. In *POPL*.

Reese, R. (2014). *Java 8 New Features: A Practical Heads-Up Guide*. P8Tech.

Robinson, A. and Lippmeier, B. (2016). Icicle: write once, run once. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing*, pages 2–8. ACM.

Robinson, A. and Lippmeier, B. (2017). Machine fusion: merging merges, more or less. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, pages 139–150. ACM.

Robinson, A., Lippmeier, B., and Keller, G. (2014). Fusing filters with integer linear programming. In *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*, pages 53–62. ACM.

Rompf, T., Sujeeth, A. K., Amin, N., Brown, K. J., Jovanovic, V., Lee, H., Jonnalagedda, M., Olukotun, K., and Odersky, M. (2013). Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *ACM SIGPLAN Notices*. ACM.

Shaikhha, A., Dashti, M., and Koch, C. (2018). Push versus pull-based loop fusion in query engines. *Journal of Functional Programming*, 28.

Shivers, O. (2005). The anatomy of a loop: a story of scope and control. In *ACM SIGPLAN Notices*. ACM.

Snoyman, M. (2011). The conduit Haskell package. `http://hackage.haskell.org/package/conduit`.

Song, Y., Xu, R., Wang, C., and Li, Z. (2004). Improving data locality by array contraction. *Computers, IEEE Transactions on*.

Soule, R., Gordon, M. I., Amarasinghe, S., Grimm, R., and Hirzel, M. (2013). Dynamic expressivity with static optimization for streaming languages. In *The 7th ACM International Conference on Distributed Event-Based Systems*.

Stephens, R. (1997). A survey of stream processing. *Acta Informatica*.

Stuijk, S., Geilen, M., Theelen, B., and Basten, T. (2011). Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Embedded Computer Systems (SAMOS), International Conference on*.

Svenningsson, J. (2002). Shortcut fusion for accumulating parameters & zip-like functions. In *ACM SIGPLAN Notices*.

Svensson, B. J. and Svenningsson, J. (2014). Defunctionalizing push arrays. In *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*.

Thies, W., Karczmarek, M., and Amarasinghe, S. (2002). StreamIt: A language for streaming applications. In *Compiler Construction*.

Thompson, M. (2015). The streaming Haskell package. `http://hackage.haskell.org/package/streaming`.

Van Kampenhout, R., Stuijk, S., and Goossens, K. (2015). A scenario-aware dataflow programming model. In *Digital System Design (DSD), Euromicro Conference on*.

Venkat, A., Shantharam, M., Hall, M. W., and Strout, M. M. (2014). Non-affine extensions to polyhedral code generation. In *CGO: Code Generation and Optimization*.

Vrba, Z., Halvorsen, P., Griwodz, C., and Beskow, P. (2009). Kahn process networks are a flexible alternative to MapReduce. In *High Performance Computing and Communications, 2009. HPCC'09. 11th IEEE International Conference on*, pages 154–162. IEEE.

Wadler, P. (1984). Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 45–52. ACM.

Wadler, P. (1990). Deforestation: Transforming programs to eliminate trees. *Theoretical computer science*, 73(2):231–248.

Welford, B. (1962). Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420.

FIGURES

TABLES

LISTINGS

CHAPTER A

BENCHMARK CODE

This appendix includes the implementations of benchmark programs that were mentioned previously in Chapter 5, but were not shown there.

```
priceAnalysesPipes (fpStock,fpMarket) =
  (,) <$> priceOverTime fpStock <*> priceOverMarket fpStock fpMarket

priceOverTime fpStock =
  Fold.purely P.fold Stats.regressionCorrelation $ go
 where
  go
   = sourceRecords fpStock
   P.>-> P.map (λs -> (daysSinceEpoch $ time s, cost s))

priceOverMarket :: FilePath -> FilePath -> IO Double
priceOverMarket fpStock fpMarket =
  Fold.purely P.fold Stats.regressionCorrelation $ go
 where
  go
   = joinBy (λs m -> time s `compare` time m)
        (sourceRecords fpStock)
        (sourceRecords fpMarket)
   P.>-> P.map (λ(s,m) -> (cost s, cost m))
```

Listing A.1: Pipes two-pass implementation of `priceAnalyses`

```
priceAnalysesStreaming (fpStock,fpMarket) = do
  (pom S.:> (pot S.:> (),_)) ← priceOverMarket
      (S.store priceOverTime $ sourceRecords fpStock)
      (sourceRecords fpMarket)
  return (pot,pom)

priceOverTime stock
 = Fold.purely S.fold covariance
 $ S.map (λs → (daysSinceEpoch (time s), cost s)) stock

priceOverMarket stock market
 = Fold.purely S.fold covariance
 $ S.map (λ(s,m) → (cost s, cost m))
 $ joinBy (λs m → time s `compare` time m)
   stock market
```

Listing A.2: Streaming implementation of `priceAnalyses`

```
quickhull :: (Vector Point → IO (Point,Point))
          → (Line → Vector Point → IO (Point, Vector Point))
          → Vector Point
          → IO (Vector Point)
quickhull fPivots fFilterMax ps0
 | null ps0 =
   return empty
 | otherwise = do
   (l,r) ← fPivots ps0
   top   ← go l r ps0
   bot   ← go r l ps0
   return (singleton l ++ top ++ singleton r ++ bot)
 where
  go l r ps
   | null ps =
     return empty
   | otherwise = do
     (pt,above) ← fFilterMax (l,r) ps
     left       ← go l pt above
     right      ← go pt r above
     return (left ++ singleton pt ++ right)
```

Listing A.3: Quickhull skeleton parameterised by filterMax and pivots

```
filterMaxHand l ps
 | Unbox.length ps == 0
 = return ((0,0), Unbox.empty)
 | otherwise = do
   mv ← MUnbox.unsafeNew $ Unbox.length ps
   (x,y,wix) ← go0 mv
   v ← Unbox.unsafeFreeze $ MUnbox.unsafeSlice 0 wix mv
   return ((x,y), v)
 where
  {-# INLINE go0 #-}
  go0 !mv = do
   let (x0,y0) = Unbox.unsafeIndex ps 0
   let d0 = distance (x0,y0) l
   case d0 > 0 of
    True → do
      MUnbox.unsafeWrite mv 0 (x0,y0)
      go mv 1 1 x0 y0 d0
    False → do
     go mv 1 0 x0 y0 d0

  {-# INLINE go #-}
  go !mv !ix !writeIx !x1 !y1 !d1
   = case ix ≥ Unbox.length ps of
      True → return (x1,y1, writeIx)
      False → do
       let (x2,y2) = Unbox.unsafeIndex ps ix
       let d2 = distance (x2,y2) l
       case d2 > 0 of
        True → do
          MUnbox.unsafeWrite mv writeIx (x2,y2)
          case d1 > d2 of
           True → go mv (ix + 1) (writeIx + 1) x1 y1 d1
           False → go mv (ix + 1) (writeIx + 1) x2 y2 d2
        False →
          case d1 > d2 of
           True → go mv (ix + 1) writeIx x1 y1 d1
           False → go mv (ix + 1) writeIx x2 y2 d2
```

Listing A.4: Hand-fused implementation of `filterMax`

```
filterMaxVectorShare l ps
 = let annot = Unbox.map (λp → (p, distance p l)) ps
       point = fst
             $ Unbox.maximumBy (compare `on` snd) annot
       above = Unbox.map fst
             $ Unbox.filter ((>0) ∘ snd) annot
   in return (point, above)
```

Listing A.5: Vector / share implementation of `filterMax`

```
filterMaxVectorRecompute l ps
 = let annot1 = Unbox.map (λp → (p, distance p l)) ps
       point  = fst
              $ Unbox.maximumBy (compare `on` snd) annot1
       annot2 = Unbox.map (λp → (p, distance p l)) ps
       above  = Unbox.map fst
              $ Unbox.filter ((>0) ∘ snd) annot2
   in return (point, above)
```

Listing A.6: Vector / recompute implementation of `filterMax`

```
filterMaxPipes l ps = do
  r  ← MUnbox.unsafeNew (Unbox.length ps)
  ix ← newIORef 0
  pt ← newIORef (0,0)
  P.runEffect (sourceVector ps        P.>→
               annot                  P.>→
               filterAndMax r ix pt 0 (0,0) (-1/0))
  pt' ← readIORef pt
  ix' ← readIORef ix
  r'  ← Unbox.unsafeFreeze $ MUnbox.unsafeSlice 0 ix' r
  return (pt', r')
 where
  annot = P.map (λp → (p, distance p l))

  filterAndMax !vecR !ixR !ptR !ix (!x,!y) !d1 = do
    lift $ writeIORef ixR ix
    lift $ writeIORef ptR (x,y)
    (p2,d2) ← P.await
    let (!p',!d') = if d1 > d2 then ((x,y),d1) else (p2,d2)
    case d2 > 0 of
     True → do
      lift $ MUnbox.unsafeWrite vecR ix p2
      filterAndMax vecR ixR ptR (ix+1) p' d'
     False → do
      filterAndMax vecR ixR ptR ix p' d'
```

Listing A.7: Pipes implementation of `filterMax`

```
compressorVector :: Vector Double → IO (Vector Double)
compressorVector ins = do
  let squares = Unbox.map        (λx → x * x) ins
  let avg     = Unbox.postscanl' lop   0       squares
  let mul     = Unbox.map        clip          avg
  let out     = Unbox.zipWith    (*)   mul     ins
  return out
```

Listing A.8: Vector implementation of `compressor`

```
compressorLopVector :: Vector Double → IO (Vector Double)
compressorLopVector ins = do
  let lopped  = Unbox.postscanl' lop20k 0 xs
  let squares = Unbox.map (λx → x * x) lopped
  let avg     = Unbox.postscanl' expAvg 0 squares
  let root    = Unbox.map clipRoot avg
  let out     = Unbox.zipWith (*) root lopped
  return out
```

Listing A.9: Vector implementation of `compressor` with low-pass

```
append2Conduit in1 in2 out =
  C.runConduit (sources C..| sinks)
 where
  sources = sourceFile in1 >> sourceFile in2

  sinks = do
   (i,_) ← C.fuseBoth (counting 0) (sinkFile out)
   return i

  counting i = do
   e ← C.await
   case e of
    Nothing   → return i
    Just v → do
     C.yield v
     counting (i + 1)
```

Listing A.10: Conduit implementation of `append2`

```
append2Pipes in1 in2 out = do
  h  ← IO.openFile out IO.WriteMode
  i  ← P.runEffect $ go h
  IO.hClose h
  return i
 where
  go h =
   let ins  = sourceFile in1 >> sourceFile in2
       ins' = counting ins 0
       outs = sinkHandle h
   in ins' P.>→ outs

  counting s i = do
   e ← P.next s
   case e of
    Left _end → return i
    Right (v,s') → do
     P.yield v
     counting s' (i + 1)
```

Listing A.11: Pipes implementation of append2

```
append2Hand in1 in2 out = do
  f1 ← IO.openFile in1 IO.ReadMode
  f2 ← IO.openFile in2 IO.ReadMode
  h  ← IO.openFile out IO.WriteMode
  i  ← go1 h f1 f2 0

  IO.hClose f1
  IO.hClose f2
  IO.hClose h
  return i
 where
  go1 h f1 f2 lns = do
   f1' ← IO.hIsEOF f1
   case f1' of
    True → go2 h f2 lns
    False → do
     l ← Char8.hGetLine f1
     Char8.hPutStrLn h l
     go1 h f1 f2 (lns + 1)

  go2 h f2 lns = do
   f2' ← IO.hIsEOF f2
   case f2' of
    True → return lns
    False → do
     l ← Char8.hGetLine f2
     Char8.hPutStrLn h l
     go2 h f2 (lns + 1)
```

Listing A.12: Hand-fused implementation of append2

```
append2Streaming in1 in2 out = do
  sinkFile out $ go (sourceFile in1) (sourceFile in2)
 where
  go s1 s2 = S.store S.length_ $ (s1 >> s2)
```

Listing A.13: Streaming implementation of append2

```
part2Pipes in1 out1 out2 = do
  o1 ← IO.openFile out1 IO.WriteMode
  o2 ← IO.openFile out2 IO.WriteMode
  ref ← newIORef (0,0)
  P.runEffect (sourceFile in1 P.>→ go ref o1 o2 0 0)
  IO.hClose o1
  IO.hClose o2
  readIORef ref
 where

  go ref o1 o2 !c1 !c2 = do
   lift $ writeIORef ref (c1, c2)
   v ← P.await
   case () of
    _
     | prd v → do
      lift $ Char8.hPutStrLn o1 v
      go ref o1 o2 (c1 + 1) c2
     | otherwise → do
      lift $ Char8.hPutStrLn o2 v
      go ref o1 o2 c1 (c2 + 1)

  prd l = ByteString.length l `mod` 2 == 0
```

Listing A.14: Pipes implementation of `part2`

```
part2Hand in1 out1 out2 = do
  f1 ← IO.openFile in1 IO.ReadMode
  o1 ← IO.openFile out1 IO.WriteMode
  o2 ← IO.openFile out2 IO.WriteMode
  r ← go f1 o1 o2 0 0
  IO.hClose f1
  IO.hClose o1
  IO.hClose o2
  return r
 where
  go i1 o1 o2 c1 c2 = do
   i1' ← IO.hIsEOF i1
   case i1' of
    True → return (c1, c2)
    False → do
     l ← Char8.hGetLine i1
     case ByteString.length l `mod` 2 == 0 of
      True  → do
        Char8.hPutStrLn o1 l
        go i1 o1 o2 (c1 + 1) c2
      False → do
        Char8.hPutStrLn o2 l
        go i1 o1 o2 c1 (c2 + 1)
```

Listing A.15: Hand implementation of `part2`

```
partitionAppendV2Loop :: Unbox.Vector Int → IO (Unbox.Vector Int)
partitionAppendV2Loop !xs = do
  let (evens,odds) = Unbox.partition (λi → i `mod` 2 == 0) xs
  let evens'       = Unbox.map       (λi → i `div` 2)      evens
  let odds'        = Unbox.map       (λi → i * 2)          odds
  let apps         = evens' Unbox.++ odds'
  return apps

partitionAppendV2Source :: Unbox.Vector Int → IO (Unbox.Vector Int)
partitionAppendV2Source !xs = do
  let p i          = i `mod` 2 == 0
  let evens        = Unbox.filter        p          xs
  let odds         = Unbox.filter (not ∘ p)         xs
  let evens'       = Unbox.map    (λi → i `div` 2) evens
  let odds'        = Unbox.map    (λi → i * 2)     odds
  let apps         = evens' Unbox.++ odds'
  return apps
```

Listing A.16: Vector implementations of `partitionAppend`