# 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

September 3, 2018

# Thesis/Dissertation Sheet

| | | |
|---|---|---|
| Surname/Family Name | : | **Robinson** |
| Given Name/s | : | **Amos Stephen** |
| Abbreviation for degree as in the University calendar | : | **PhD** |
| Faculty | : | **Faculty of Engineering** |
| School | : | **School of Computer Science and Engineering** |
| Thesis Title | : | **The stuff that streams are made of** |

**Abstract 350 words maximum: (PLEASE TYPE)**

To learn interesting things from large datasets, we generally want to perform lots of queries. If we compute each query separately, we may spend more time reading the data than we spend computing the answer. Instead of computing 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 at a time are push streams and Kahn process networks.

Push streams can execute multiple queries at a time, but these queries can be unwieldy to write as they 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 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 both multiple inputs and multiple queries, but require dynamic scheduling and inter-process communication, both of which 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 synchronised product of processes, which is generally used as a proof technique rather than an optimisation.

Declaration relating to disposition of project thesis/dissertation

I hereby grant to the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or in part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all property rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstracts International (this is applicable to doctoral theses only).

…………………………………………………………… | ……………………………………….. | 2018/08/24
…… | ……………… | …………………………………..……….
Signature | Witness Signature | Date

The University recognises that there may be exceptional circumstances requiring restrictions on copying or conditions on use. Requests for restriction for a period of up to 2 years must be made in writing. Requests for a longer period of restriction may be considered in exceptional circumstances and require the approval of the Dean of Graduate Research.

| **FOR OFFICE USE ONLY** | Date of completion of requirements for Award: |
|---|---|

**ORIGINALITY STATEMENT**

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed  ……………………………………………...............

Date  2018/08/24
……………………………………………...............

# CONTENTS

# ABSTRACT

To learn interesting things from large datasets, we generally want to perform lots of queries. If we compute each query separately, we may spend more time reading the data than we spend computing the answer. Instead of computing 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 at a time are push streams and Kahn process networks.

Push streams can execute multiple queries at a time, but these queries can be unwieldy to write as they 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 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 both multiple inputs and multiple queries, but require dynamic scheduling and inter-process communication, both of which 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 synchronised product of processes, which is generally used as a proof technique rather than an optimisation.

# F I G U R E S

# T A B L E S

# L I S T I N G S

# 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 large datasets, we generally want to perform lots of queries. When our query is simple and our data is big, we might spend more time reading the data than we spend computing the answer. In this case, 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), which we can write by composing stream transformers together. Composing stream transformers together can add some performance overhead, which is usually removed by *fusing* together 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 execute multiple queries at a time, but these queries 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.

However, push streams do not support computations with multiple inputs except for non-deterministically merging two streams, and in some circumstances appending streams, as we shall see in section 2.3. As an alternative to push streams, Kahn process networks support both multiple inputs and multiple queries, but require dynamic scheduling and inter-process communication, both of which introduce significant runtime overhead. In section 4.3 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 (section 6.1) and demonstrates the connection between fusion and synchronised product of processes (section 6.2), which is generally used as a proof technique rather than an optimisation.

## 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 queries, and user-defined combinators. In this streaming model, each combinator in each query is implemented as a sequential process, and together, the combinators of all queries form a concurrent process network. Processes are

then fused together using an extension of synchronised product. The fusion algorithm is described in chapter 4.

FORMAL PROOF OF CORRECTNESS OF FUSION:  a proof of correctness for the above-mentioned fusion system, 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.

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

In this thesis, we 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 a particular index. These restrictions mean that a streaming program cannot, for example, sort all the input data in a single pass, 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; in this thesis we focus on 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 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 connected together to create larger programs. The benefit of this style is that each stream transformer can be reasoned about and tested in isolation with no hidden dependencies between stream 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 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 different streaming models 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 start by describing a situation in which we would like to execute many queries. 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 ignore details about efficiency. 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:

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

We wish to evaluate 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 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.
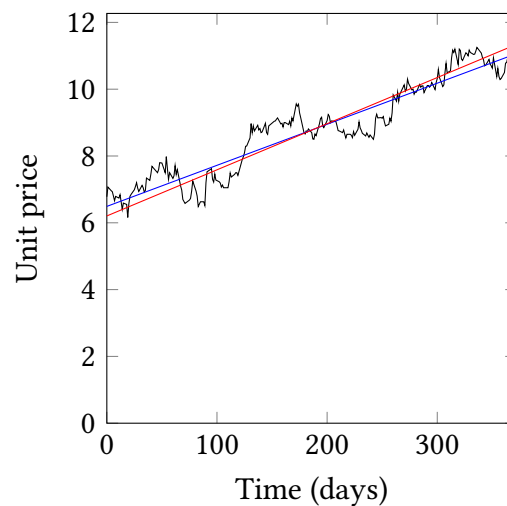


Figure 2.1: Analysis of a stock over a year

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

```haskell
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')
     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 2.1: One-pass correlation implementation

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

We can implement a one-pass correlation algorithm, and although the details are quite complicated, we can express it 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 2.1 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 fold 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 `map`. 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. In the comparison of stock price to index price, we can see that the stock has grown faster than the index.

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

(a) Market over time
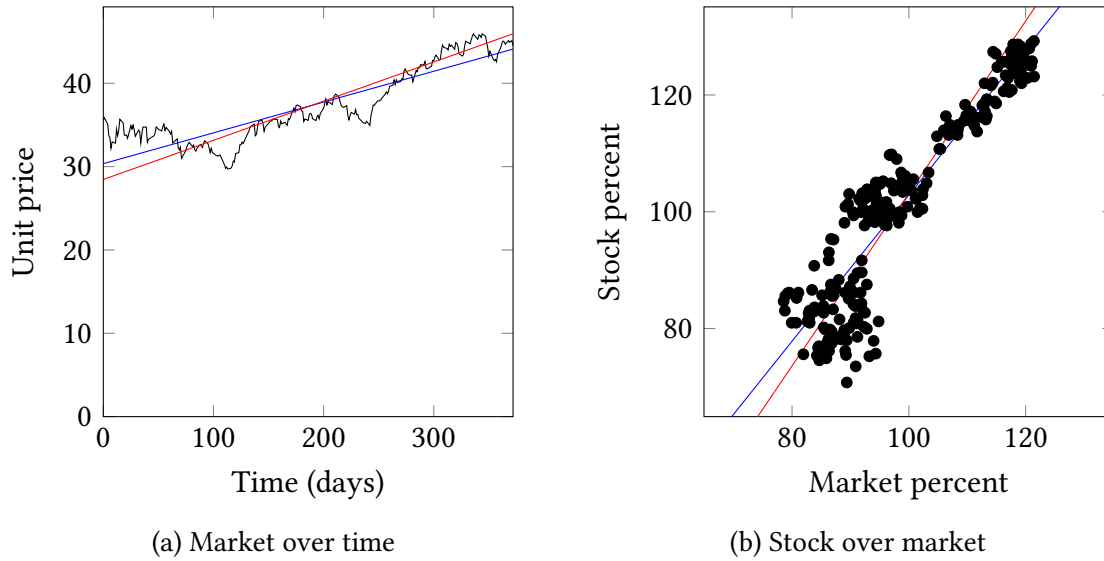
(b) Stock over market

Figure 2.2: Analysis of a stock compared to market index

```
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)
```

To match each stock day against the corresponding market index day, and discard any days missing from either input, we join the stock with the index based on the date. 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:
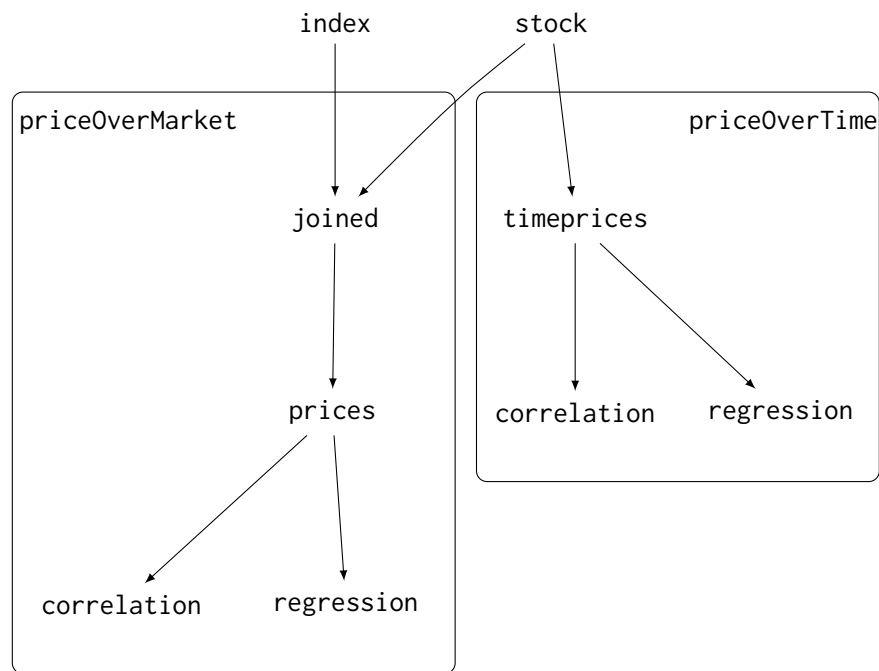
Figure 2.3: Dependency graph for queries `priceOverTime` and `priceOverMarket`

```
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 all queries. The nodes in this graph are the two input lists `stock` and `index`, each intermediate list, and the `correlation` and `regression` functions which summarise the list values. The edges are dependencies from one value to another; `joined` is computed by joining together the `stock` and `index` lists, so there are arrows from both `stock` and `index` to `joined`. 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*, which are also sometimes called iterators or cursors. The essence of a pull stream is that a consumer can *pull* on it to ask for the next value. We represent a pull stream as a function with no parameters which either returns a value, or returns Nothing when the stream is finished. Since the stream might want to read from a file or update some local state, the function is wrapped in IO:

```
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 like so:

```
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` and asks for the next value, `pull_b` in turn calls `pull_a` asking for the next value. When the stream is not finished, we apply the transform function `a_to_b` to the pulled element and return 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 `foldl`. Because pull streams can perform effects such as reading from a file, the result type for `foldl` is now wrapped in `IO`:

```
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.

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 references 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 input one after another, they would get different elements of the stream.

```haskell
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 2.2: Pull stream combinators

Listing 2.2 shows the type signatures of the push stream versions of `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.

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, pull streams, as well as many other streaming models, require that streams are not used multiple times. In fact, in section 2.4 we shall see a streaming model which is more strict and requires that streams are used exactly once (*linearly*).

Some streaming representations allow streams to be *rewinded* so they may be read multiple times from the start. When a stream is read multiple times, all the effects and all the work that went into computing the stream the first time must be done a second time. Rewinding would allow this program to compute the correct result, but the file would be read from disk again.

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 seed 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. 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 6.5.

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*

This pull stream representation can incur some streaming overhead because of the allocated Maybe values. For simple combinators such as map, however, the overhead can be optimised

away. Consider the following function, which applies two 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 `bs` and `cs` 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 value and returns it to `pull_b`. This `Just` 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 when `pull_a` returns a `Just`, `pull_b` also returns a `Just`.

To take advantage of this 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, the two maps are combined into one, removing the overhead of additional constructors. Optimising compilers perform similar transforms as part of their suite of general purpose optimisations. In the pull stream representation used here, the `filter` combinator is expressed as a recursive function, which makes it harder to inline the definition and remove the overhead. In section 6.1 we discuss more sophisticated representations; for example in the Stream Fusion (Coutts et al., 2007) representation, `filter` is defined non-recursively to assist inlining. These more sophisticated representations of pull streams 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 and improved constant factors.

## 2.3   PUSH STREAMS

*Push streams* are the conceptual dual of pull streams: rather than the consumer trying to pull from the producer, in push streams the producer pushes 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 we have a value we call push. When the stream is finished we call 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 pull streams results in a fundamental difference in how we program with push streams, and what we can express with push streams.

Although we use a slightly different representation, the push streams described here are analogous to the *sinks* described in Bernardy and Svenningsson (2015) and Lippmeier et al. (2016). In this thesis, we use the push/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. For now, we are interested in executing multiple queries; we defer further discussion of the Neumann push model to section 6.6.

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 the input to a function, making the stream element contravariant. 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 value into the returned stream, the push_a function converts this to a value of type b and pushes it further on to the consumer of b. Unlike with pull streams, a push consumer has no way of choosing among multiple inputs. 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 stream result rather than the input elements:

```
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 in IO 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 seed 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 before, 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.

```
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 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 there are no more input pairs, the stream results are paired together and flow from the argument streams to the returned stream. This inverted control flow is because the stream elements are contravariant and 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. We shall see more examples of push programs in chapter 3.

We cannot implement `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 combinator chooses which stream to pull from, pulling on 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, always ready to accept values as they come.

This inability to join two streams by date is a symptom 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.

## 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 multiple inputs; pull streams support multiple inputs, but they do not support multiple queries (Kay, 2009). Combining pull and push streams 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 pull stream which, when pulled on, pulls from its source push_a, pushes the value to sink push_b, and returns the value to the caller. 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 with 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:

```
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)
```

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 signature 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 convert a pull stream to a push stream: we call this operation *draining* the pull stream. To drain a stream, we loop over all the values 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)
```

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 2.3 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 for join_ioo is given in listing 2.4. 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

```haskell
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 2.3: Polarised implementation of join_iii

```
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 2.4: Polarised implementation of `join_ioo`

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 the two `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 value or the result. Unfilled circles ○ represent push streams because they are a hole which values can be pushed into.

We begin by classifying both input streams as pull streams: we can convert pull streams into push streams but not vice versa. The `stock` input stream is used twice, so at least one of the use-sites must be push. Since we were able to express `priceOverTime` entirely as a push stream, we duplicate `stock` into a push stream for `priceOverTime` and a pull stream for
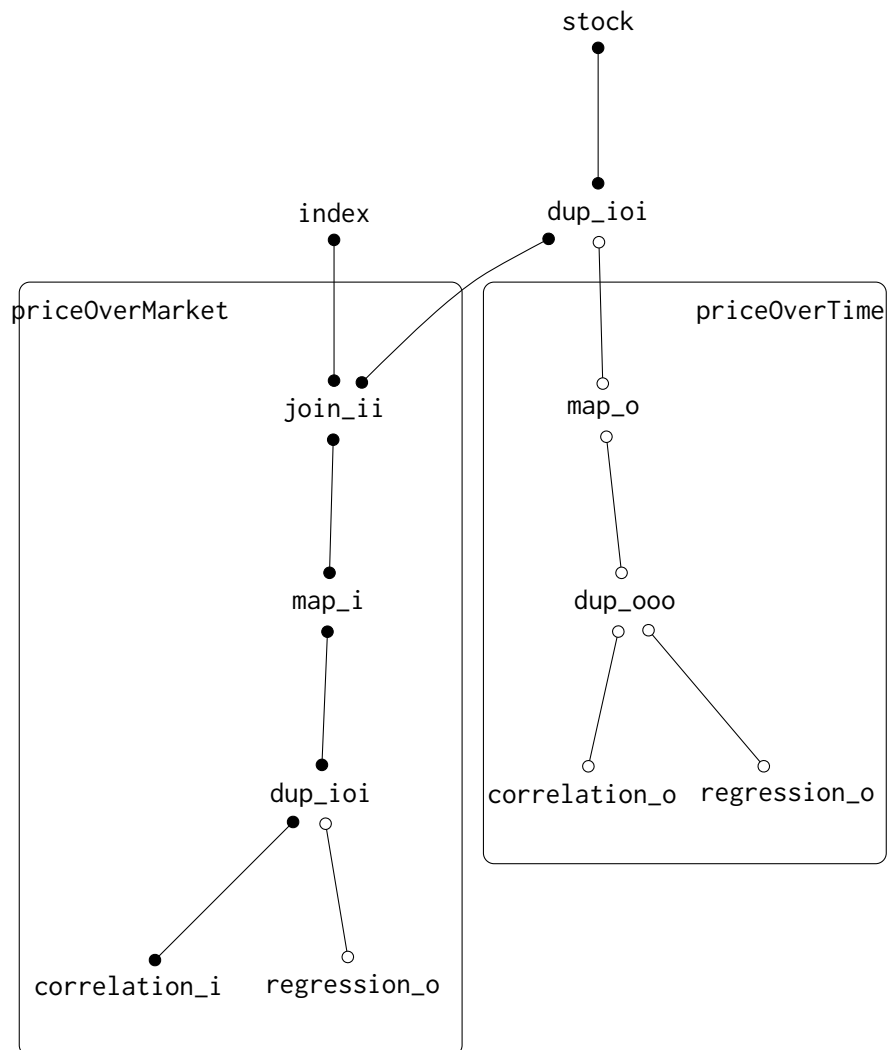
Figure 2.4: Polarised dependency graph for priceOverTime and priceOverMarket

`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 ways to assign polarities to this program, but they all compute the same result.

Listing 2.5 shows the implementation of this polarised dependency graph for `priceOverTime` and `priceOverMarket`. 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 requires pattern-matching on the nested tuple.

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 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, 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 definition of `correlation` takes a list of pairs of doubles and performs a fold over them. We could have also written `correlation` to take two lists of doubles and pairing

```
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 2.5: Polarised implementation of priceOverTime and priceOverMarket
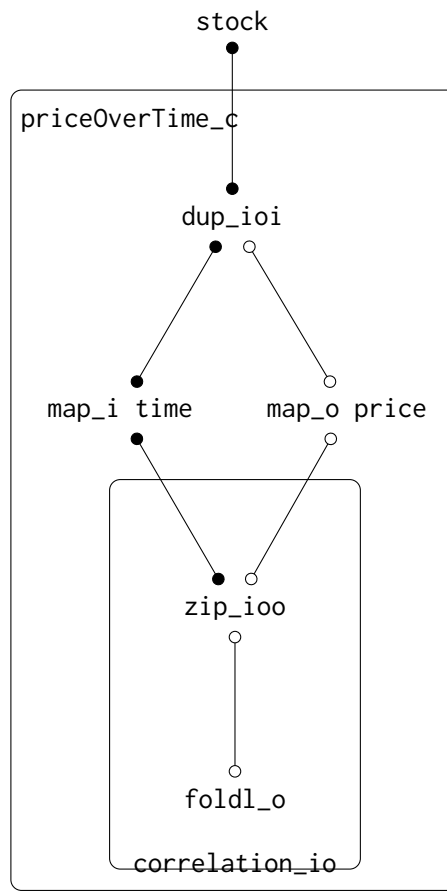
Figure 2.5: Polarised dependency graph with diamond

the elements together before folding them. 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 the difference is usually minor.

With polarised streams, we cannot execute `priceOverTime` with the pair-of-lists version, although we can assign polarities. Figure 2.5 shows a polarised graph for a version of `priceOverTime` that only computes the correlation and uses zip. The `zip_ioo` combinator, implemented in listing 2.6, 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.

```
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 2.6: Polarised implementation of `zip_ioo`

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. With downward arrows for pull streams and upward arrows for push streams, there is a cycle between `dup_ioi`, `zip_ioo` and the two maps. This cycle complicates translating the graph to an implementation.

The incomplete implementation in listing 2.7 also demonstrates the recursive dependency between `stock'`, `times`, `prices` and `cor`. 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 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

```
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 2.7: Incomplete polarised implementation of `priceOverTime_c`

flow for push streams is in 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 in where the computation is controlled. In pull streams, the consumer controls the computation. In push streams, the producer controls the computation. In polarised streams, the active combinator, which may be a pull consumer and a push producer, controls the computation. All these systems have one place controlling the computation. When we have multiple queries to execute, it can be hard to choose just one combinator to control the entire computation.

An alternate streaming model, which allows many places 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 which they can pull values from, and output channels which they can push values to. 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 how processes are interleaved during execution, depends on the environment. 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 that ensures that processes compute the same result across different schedules (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. All communication between processes is through first-in-first-out channels. Processes with shared mutable state can non-deterministically compute different results: if one process were reading from mutable state while another were writing a new value, the reading process may get the old value or new value, depending on how the processes were scheduled. Restricting all communication to go through channels outlaws processes from communicating via shared mutable state. Reading from channels is blocking: processes are not allowed 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 allowed to push to a channel: if two

```haskell
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 2.8: Types and combinators for Kahn process networks

processes were able to push to the same channel at the same time, the scheduler would have to decide the order the 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 also convert both push streams and pull streams to processes.

Kahn process networks can use bounded channels to ensure communication between processes executes in bounded memory. Kahn process networks with bounded channels still compute results deterministically, but can introduce *artificial deadlocks* when 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 2.8 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; 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.

Listing 2.9 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. The conversion here from list to process network is almost a purely syntactic transform, 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 a `Maybe` value, which can sometimes be removed by general purpose compiler optimisations. 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 must generally lock the communication channel to ensure it is the only process with access to the channel before copying the value into a buffer where it can be read by the other process. Concurrent process network implementations often amortise the cost of communication by *chunking* messages together: instead of sending many messages with one value in each, chun-

```
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 2.9: Implementation of `priceAnalyses` queries as a Kahn process network

ked 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 are less likely to fit in cache. Since each channel between 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 example are desirable because they allow us to write a process for 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 determined by synchronisation and scheduling overheads (Chen et al., 1990). We can reduce the amount of communication by combining multiple connected processes together into one larger process. The combined process performs the task of multiple individual processes, but communicates by local variables instead of channels. In chapter 4 we describe an algorithm to combine processes together to reduce overhead. For `priceAnalysis`, our algorithm can combine all the processes together into a single processes. A single process executes sequentially; fusing the entire network into a single process removes any potential speedup from 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 multiple inputs, and can take advantage of an optimising compiler to reduce overhead. Push streams support multiple 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 (chapter 3). 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).

# 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 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);

---

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

- We present a fold-based intermediate language, which allows the query fusion trans-
  formation 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).

## 3.1   GOLD PANNING WITH ICICLE

In 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. As Icicle uses push streams, the queries
we write have a single input stream of the prices for a particular company; Icicle cannot express
the `priceOverMarket` example, which contains multiple input streams. Suppose we want to
compute the number of days where the open price exceeded the close price, and vice versa.
We also want 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 Ints. In Icicle, input tables have an implicit time field and the input stream is
sorted chronologically.

We can express the same three queries using the push streams from section 2.3, as in the
following program:

```haskell
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)
```

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 with the same filter predicate. When the same computation is used by multiple queries, we would 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 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 and 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. 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:

```
table kvs { key : Date; value : Int }
query avg = 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. Our implementation is available at `https://github.com/amosr/icicle`. As of 2018, this implementation is still running in production in the machine-learning pipeline at Ambiata.

## 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 avg = 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 (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` is selecting 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 avg = 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 lookup 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` $\tau$ are taken from the input stream on a per-element

basis, whereas values of type `Aggregate` $\tau$ 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, which 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 the unstreamable query from section 3.2 were converted to LUSTRE syntax then it 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 do this by remembering the values of all intermediate aggregates that were computed in the query, and updating them as new data arrives. In the avg 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 which 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 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.

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

$$T \qquad ::= \quad \mathtt{Int} \mid \mathtt{Bool} \mid \mathtt{Map}\ T\ T \mid (T\ \times\ T)$$

$$M \qquad ::= \quad T \mid \mathtt{Element}\ T \mid \mathtt{Aggregate}\ T$$

$$F \qquad ::= \quad \overline{(M)}\ \rightarrow\ M$$

$$Table \quad ::= \quad \mathtt{table}\ x\ \{\ \overline{(x\ :\ T;)}\ \}$$

$$
\begin{aligned}
Exp, e \quad ::= \ & x \mid V \mid Prim\ \overline{Exp} \mid x\ \overline{Exp} \\
\mid \ & \mathtt{let} \quad x\ =\ Exp \quad \mathtt{in}\ Exp \\
\mid \ & \mathtt{fold} \quad x\ =\ Exp\ \mathtt{then}\ Exp \\
\mid \ & \mathtt{filter} \qquad Exp \quad \mathtt{of}\ Exp \\
\mid \ & \mathtt{group} \qquad\ \ Exp \quad \mathtt{of}\ Exp
\end{aligned}
$$

$$
\begin{aligned}
Prim, p \quad ::= \ & (\mathtt{+}) \mid (\mathtt{-}) \mid (\mathtt{*}) \mid (\mathtt{/}) \mid (\mathtt{==}) \mid (\mathtt{/=}) \mid (\mathtt{<}) \mid (\mathtt{>}) \mid (,) \\
\mid \ & \mathtt{lookup} \mid \mathtt{fst} \mid \mathtt{snd}
\end{aligned}
$$

$$V, v \qquad ::= \quad \mathbb{N} \mid \mathbb{B} \mid \{V \Rightarrow V\} \mid (V\ \times\ V)$$

$$
\begin{aligned}
Def \qquad ::= \ & \mathtt{function}\ f\ \overline{(x\ :\ M)}\ =\ Exp \\
\mid \ & \mathtt{query}\ x\ =\ Exp
\end{aligned}
$$

$$Top \qquad ::= \quad Table;\ \overline{Def};$$

Figure 3.1: Icicle grammar

`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 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 $\Gamma$. 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 $\Gamma$ and function or query, and produces an environment containing the function or query name and its type. Finally, 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) assigns an expression either `Element` or `Aggregate` type.

Rules (TcPrimApp) and (TcFunApp) produce the type of a primitive or function applied to its arguments. Rule (TcLet) is standard.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3.2: Types of expressions

In rule (TcFold) the initial value has value type $T$. The type of the current element from the stream is added to the context of $e_k$ as an `Element`, and the result of the overall fold is an `Aggregate`.

Rule (TcFilter) requires the first argument of a `filter` to be (`Element Bool`), a stream of predicate flags. The second argument must be an `Aggregate`, which is a 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 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 elements from different streams can always be pairwise joined. Introducing literal stream values and representing them as a list of values would invalidate this invariant because the length of the input stream is unknown. 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 must be 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 $\Sigma$ defines a heap containing stream values. Grammar $V'$ defines the results 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 (zero) state, an update (konstrukt) meta-function to be applied to each stream element and current state, and an eject meta-function to be applied to the final state.

In the grammar $V'$, we write ($\overset{\bullet}{\to}$) 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 this data.

The judgment form ($N \mid \Sigma \;\vdash\; e \;\Downarrow\; V'$) defines a big-step evaluation relation: under evaluation mode $N$ with heap $\Sigma$, 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 already a completed result. Rule (EVar) performs variable lookup in the heap. Rule (ELet) evaluates the bound expression under the given mode.

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

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

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

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

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

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

$$\frac{\{\texttt{Aggregate} \mid \Sigma \vdash e_i \Downarrow \texttt{Fold } z_i\, k_i\, j_i\}}{\texttt{Aggregate} \mid \Sigma \vdash p\,\{e_i\} \Downarrow \texttt{Fold}\,(z_0 \times \cdots \times z_i)\,(\lambda s\,(v_0 \times \cdots \times v_i).\, k_0\, s\, v_0 \times \cdots \times k_i\, s\, v_i)\,(\lambda(v_0 \times \cdots \times v_i).\, p\,\{j_i\, v_i\})} \text{(EPrimFold)}$$

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

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

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

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

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

Figure 3.3: Evaluation rules and auxiliary grammar

Rules (EBoxStream) and (EBoxFold) lift constant values to stream results and aggregate results respectively. To lift a constant to a stream result, we produce a meta-function that always returns the value. To lift a constant 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 constant values, streams and aggregations respectively. In (EPrimStream), the result is a new stream transformer that applies the primitive to each of the elements gained from the input streams. In (EPrimFold), the result consists of new update and eject functions that get their input values by applying the update and eject functions gained by evaluating the arguments.

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 updates the appropriate element in the map, and the eject function is applied to every accumulator.

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. In the conclusion of (EFold), we pass this stream transformer a tuple $(v, s)$, where $v$ is the new accumulator and $s$ is the current element of the stream we get from the context of the overall `fold` expression. The heap used when evaluating $k$ is updated so that references to either the stream elements or new accumulator access the appropriate side of the tuple.

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

$$
\begin{array}{lll}
PlanX & ::= & x \mid V \mid PlanP \; \overline{PlanX} \mid \lambda x. \; PlanX \\
PlanP & ::= & Prim \mid \texttt{mapUpdate} \mid \texttt{mapEmpty} \mid \texttt{mapMap} \mid \texttt{mapZip} \\
PlanF & ::= & \texttt{fold} \quad\quad x \; : \; T \; = \; PlanX \text{ then } PlanX; \\
& \mid & \texttt{filter} \;\; PlanX \quad \{ \; \overline{PlanF} \; \} \\
& \mid & \texttt{group} \quad PlanX \quad \{ \; \overline{PlanF} \; \} \\
\\
Plan & ::= & \texttt{plan } x \;\; \{ \; \overline{x \; : \; T;} \; \} \\
& & \texttt{before} \;\; \{ \; \overline{x \; : \; T \; = \; PlanX;} \; \} \\
& & \texttt{folds} \quad \{ \; \overline{PlanF} \; \} \\
& & \texttt{after} \quad \{ \; \overline{x \; : \; T \; = \; PlanX;} \; \} \\
& & \texttt{return} \; \{ \; \overline{x \; : \; T \; = \; x;} \; \}
\end{array}
$$

Figure 3.4: Query plan grammar

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

The Icicle intermediate language is similar to a physical query plan for a database system. We convert each source level 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 allowed 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 stored in 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 fold update is only performed 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 fusion can have many outputs.

Before we discuss an example query plan we first define the `count` and `sum` functions used in earlier sections. Both functions are 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);
```

Each query is converted to a query plan separately. When we convert the more query, we define the count inside a filter, and use it in the return section.

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

We convert the less and mean queries similarly. For the mean query, the sum and the count are both defined inside a filter. The division is performed in the after section because it is an aggregate operation.

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


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. 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 queries are interspersed but the stages are expressed separately, we are free to rearrange each stage without affecting the other stages.

```
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 in the general case, listing 3.1 contains the push implementation of the same queries after inlining the definition 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)

```haskell
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 3.1: Push implementation of queries after inlining combinators

algorithm to remove the duplicate work from the push implementation; such algorithms are generally polynomial time. 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 simple to translate to the intermediate language; the "mean of latest" query from section 3.2 is a bit more involved. That query, again, is:

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

We first define the mean and last functions used in the query. The mean function divides the sum 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 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 the above functions into the "mean of latest" query yields the following:

```
query avg
 =    let lst = (fold l = NO_DATE then key)
     in let map = group key of
```

---

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

```
                ( (fold s = 0 then s + value)

                / (fold c = 0 then c + 1) )

  in let ret = lookup lst map

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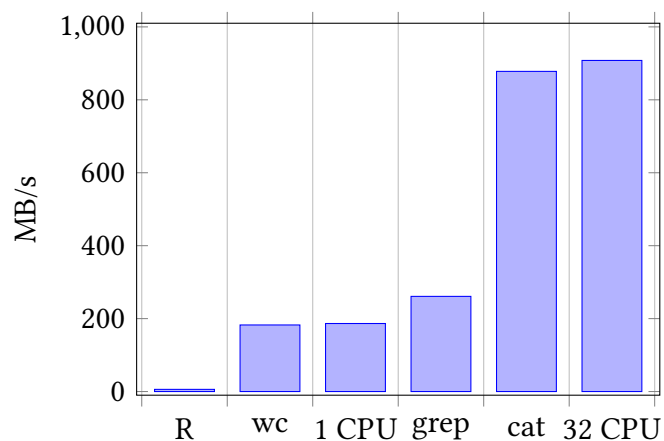
Figure 3.5: Throughput comparisons of Icicle (1 CPU and 32 CPU) against existing R code and standard Unix utilities; higher is faster.

```
after  { ret : Int = lookup lst map }
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 `filters`, the update statements are nested inside `if` statements. When translating folds nested inside `groups`, each accumulator becomes a map, and the update statements modify the element corresponding to the current key. As with `filters`, `groups` with the same key can be merged together. The nested structure of `groups` allows us to share the same set of keys across multiple accumulators, further reducing duplicate work.

## 3.4   BENCHMARKS

This section shows the results of benchmarks carried out in 2016 while working at Ambiata. 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 tens of 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. It 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 "^$"`;

- counting characters, words and lines with `wc`;

- reading and throwing away the results with `cat > /dev/null`.

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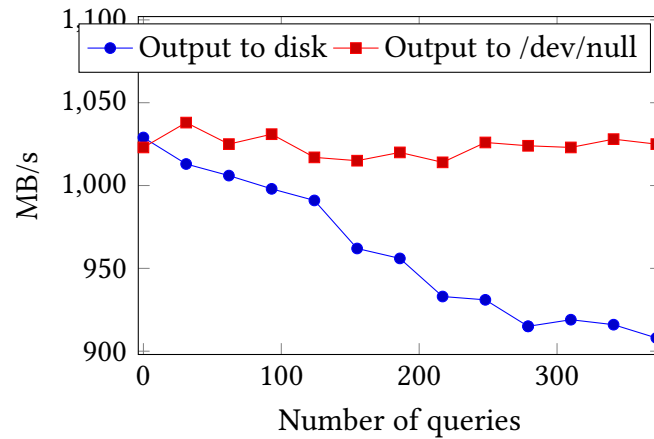
Figure 3.6: Decrease in read throughput as queries are added, comparing writing the output to disk and writing to /dev/null.

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. The time spent evaluating the queries themselves is small relative to our current IO load.

3.5 DISCUSSION

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. 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 fil-

tering 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 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 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 an algorithm for fusing pairs of processes (section 4.3);

- 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).
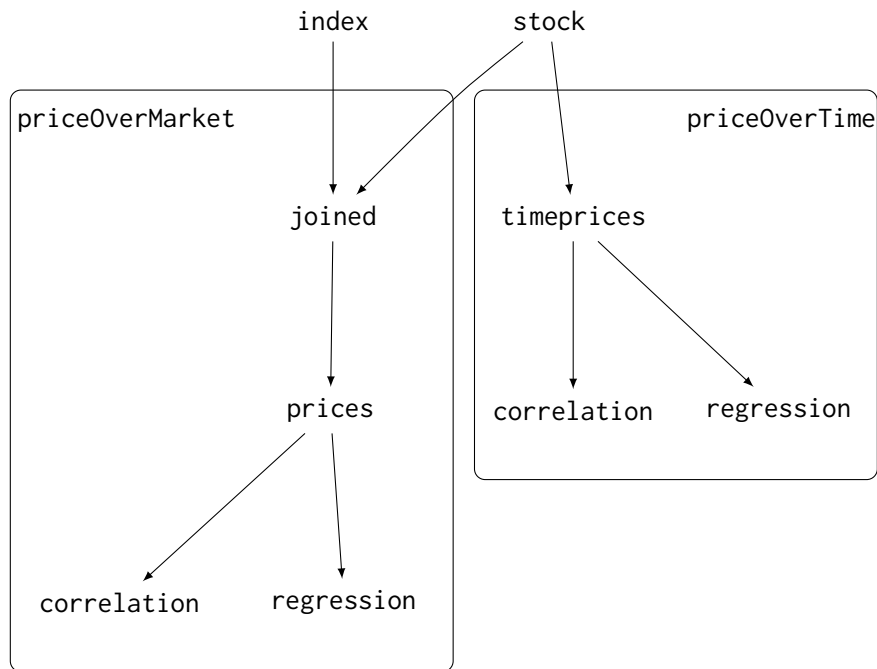
Figure 4.1: Dependency graph for priceAnalyses example

## 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.

As discussed earlier, we cannot execute this example 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 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 at the same time, 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 run-time overhead, as stream elements must be passed between processes. Instead, we wish to take this concurrent process network and convert it back to sequential code, without 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 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* (*v* (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

```
foldl
  = λ (k   : b → a → b) (z    : b) (j : b → c)
      (sIn: Stream a)      (sOut: Stream b).
    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`

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 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.

```
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 }
```

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

### 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

```
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 } }
```
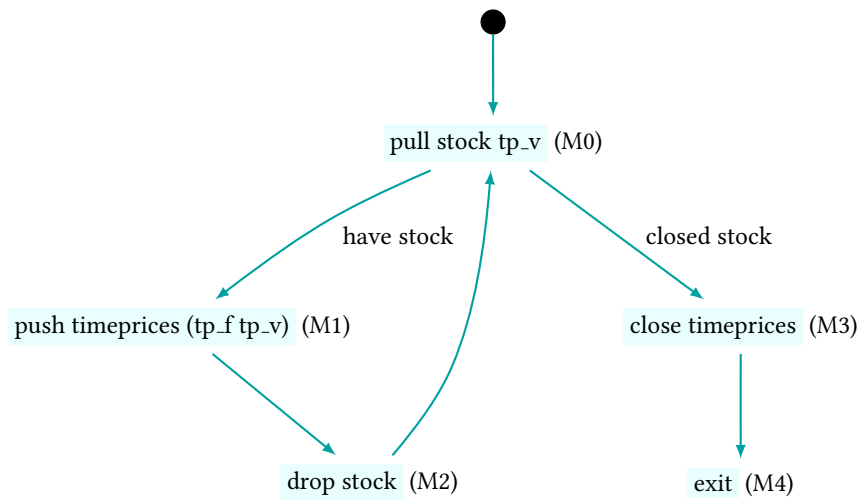


Figure 4.2: Instantiated process for map with control flow graph

```
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 } }
```
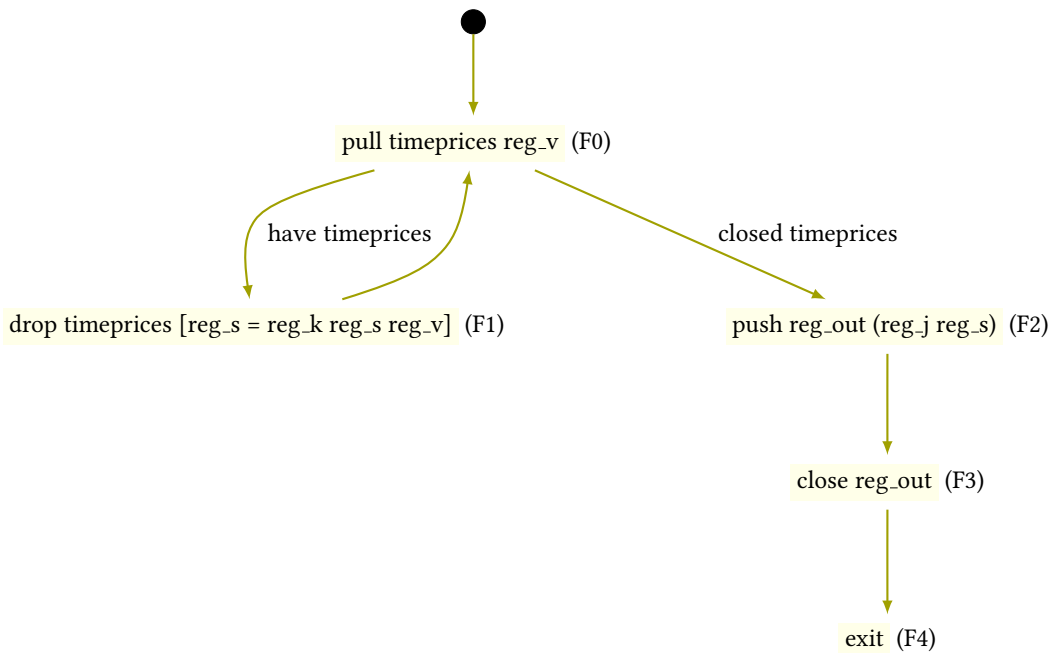


Figure 4.3: Instantiated process for fold (regression) with control flow graph

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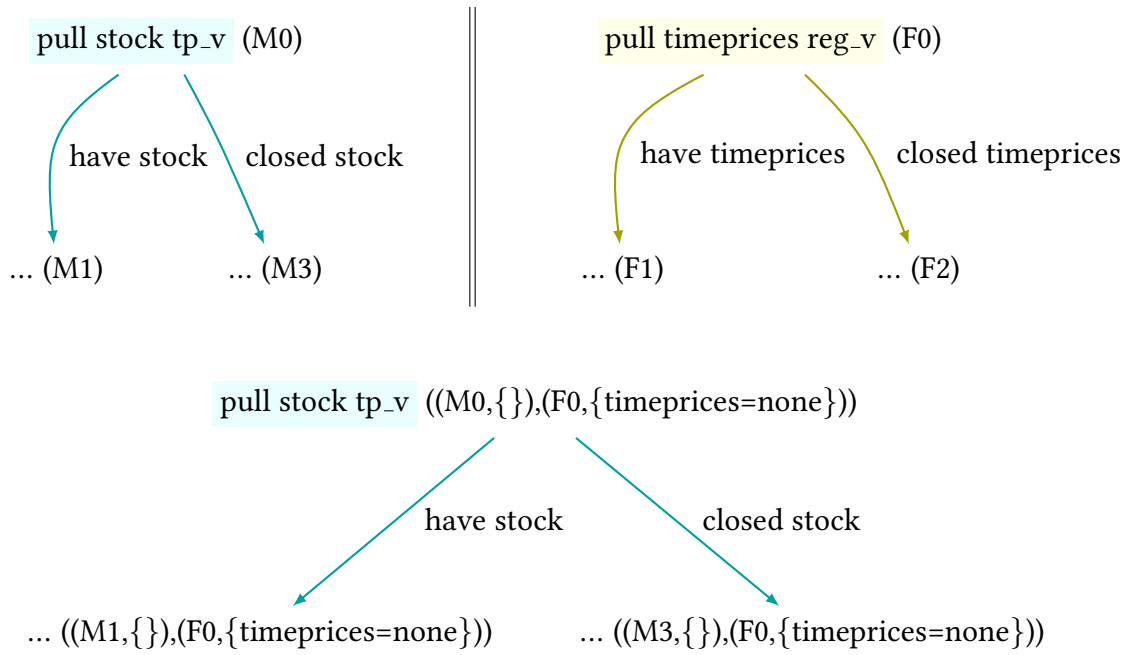
Figure 4.4: Fusing pull instructions for an unshared stream; the left process can pull from the unshared stream, while the right process must wait for the first process to produce a value

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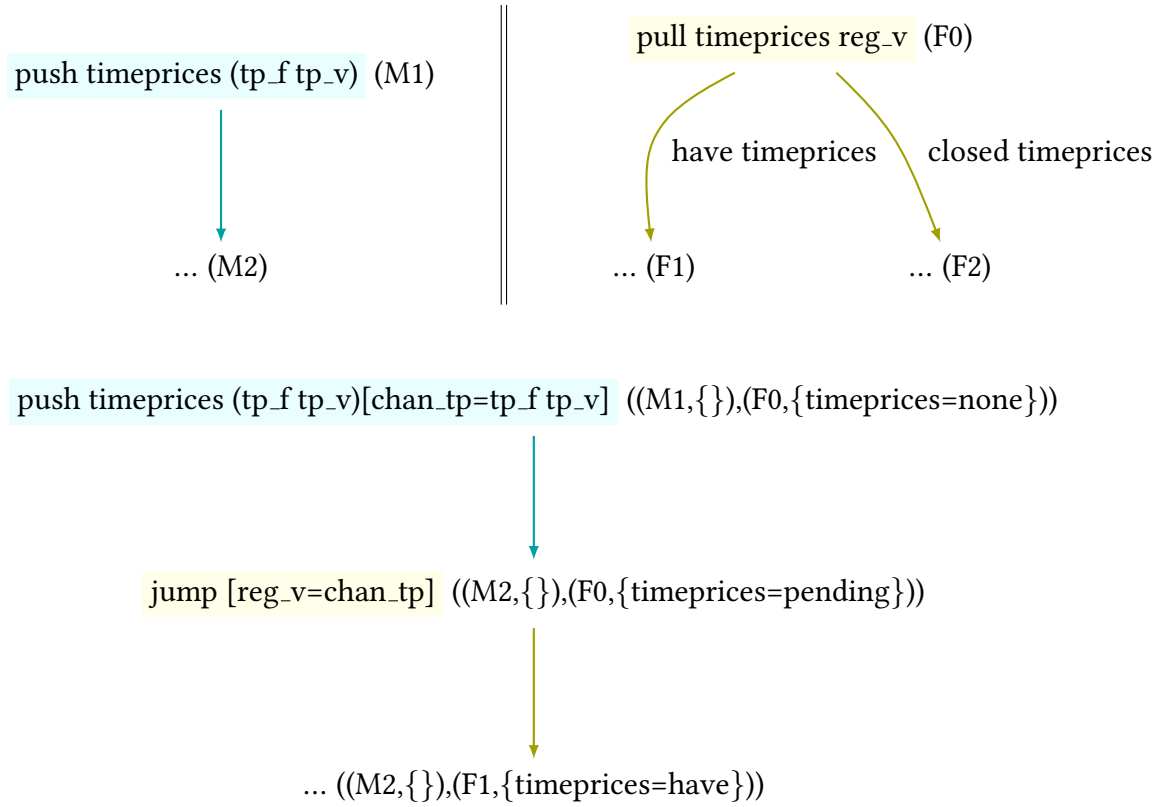
Figure 4.5: Fusing push with pull; the left process produces a value, which the right process consumes

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 4.3 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

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

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.

### 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 4.4, 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.

```
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 4.4: Process implementation of join

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 each consumer has 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 a terminated 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 network semantics so that producers do not push to terminated processes, effectively disconnecting from all inputs upon termination. However, a fused process, which performs the job of two input processes, only terminates once both input processes have terminated; as such, the fused process would only disconnect once both input 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

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

```
Process, p    ::= process
                    ins:    (Channel ↦ InputState)
                    outs:   {Channel}
                    heap:   Heap
                    label:  Label
                    instrs: (Label ↦ Instruction)
```

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

$$
\begin{array}{lll}
\textit{Variable, x} & \rightarrow & \text{(value variable)} \\
\textit{Channel, c} & \rightarrow & \text{(channel name)} \\
\textit{Label, l} & \rightarrow & \text{(label name)} \\
\textit{ChannelStates} = & & (\textit{Channel} \mapsto \textit{InputState}) \\
\textit{Action, a} & ::= & \cdot \mid \mathtt{push}\ \textit{Channel Value} \mid \mathtt{close}\ \textit{Channel}
\end{array}
$$

```
Instruction   ::= pull     Channel Variable  Next  Next
                 | push     Channel Exp        Next
                 | close    Channel            Next
                 | drop     Channel            Next
                 | case     Exp      Next       Next
                 | jump                         Next
                 | exit
```

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

Figure 4.6: Process definitions

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 the output channels; the `heap` field the process-local heap; the `label` field the label of the instruction currently being executed; and the `instrs` 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 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, blocking

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 *Variable* × *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 of a process network consists of:

1. *Injection* of an action, which is an optional stream value or stream close message, into a process or a network. Each individual process only accepts an injected value 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 gives the rules for injecting values 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.

$$\boxed{\textit{Process};\ \texttt{inject}\ \textit{Action} \Rightarrow \textit{Process}}$$

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

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

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

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

$$\boxed{\{\textit{Process}\};\ \texttt{injects}\ \textit{Action} \Rightarrow \{\textit{Process}\}}$$

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

Figure 4.7: Injection of message actions into input channels

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.

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.

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

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

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

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

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

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

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

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

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

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

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

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

Figure 4.8: Advancing processes

*Advancing*

Figure 4.8 gives the rules for advancing a single process and process networks. The statement $(i;\ is;\ bs \overset{a}{\Rightarrow} l;\ is';\ us')$ 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. As this evaluation is completely standard, we omit it to save space.

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.

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.

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

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

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

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

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

Figure 4.9: Feeding process networks

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 $o$. 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 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.

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

Figure 4.10: Fusion type definitions.

$\textit{fusePair}\ :\ \textit{Process} \rightarrow \textit{Process} \rightarrow \textit{Maybe Process}$
$\textit{fusePair}\ p\ q$
  | Just $is \leftarrow go\ \{\}\ l_0$
  = Just (process
             ins: $\ \{c \mid c = t \in cs,\ t \in \{\mathsf{in1}, \mathsf{in2}\}\}$
          outs: $\ \{c \mid c = t \in cs,\ t \in \{\mathsf{in1out1}, \mathsf{out1}\}\}$
         heap: $\ p[\mathsf{heap}] \cup q[\mathsf{heap}]$
       label: $\ l_0$
     instrs: $\ is$)
  | otherwise = Nothing
 where
 $cs\ \ = \textit{channels}\ p\ q$
 $l_0\ \ = \big(\ (p[\mathsf{label}], \{c = \mathsf{none}_F \mid c \in p[\mathsf{ins}]\})$
      $,\ (q[\mathsf{label}], \{c = \mathsf{none}_F \mid c \in q[\mathsf{ins}]\})\big)$
 $go\ bs\ (l_p, l_q)$
    | $(l_p, l_q) \in bs$
    = Just $bs$
    | Just $b \leftarrow \textit{tryStepPair}\ cs\ l_p\ p[\mathsf{instrs}][l_p]\ l_q\ q[\mathsf{instrs}][l_q]$
    = $\textit{foldM}\ go\ (bs \cup \{(l_p, l_q) = b\})\ (\textit{outlabels}\ b)$
    | otherwise = Nothing

Figure 4.11: Fusion of pairs of processes

$tryStepPair$ : ($Channel \mapsto ChannelType2$)
$\quad\quad\quad \rightarrow \quad LabelF \rightarrow Instruction \rightarrow LabelF \rightarrow Instruction$
$\quad\quad\quad \rightarrow \quad Maybe\ Instruction$
$tryStepPair\ cs\ l_p\ i_p\ l_q\ i_q\ =$
  `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` $\leftarrow i'_q$ $\quad\quad \rightarrow$ `Just` $i'_p$ $\quad\quad\quad\quad\quad$ (DeferExit1)
  | `exit` $\leftarrow i'_p$ $\quad\quad \rightarrow$ `Just` ($swaplabels\ i'_q$) (DeferExit2)
  | `jump _` $\leftarrow i'_p$ $\quad \rightarrow$ `Just` $i'_p$ $\quad\quad\quad\quad\quad$ (PreferJump1)
  | `jump _` $\leftarrow i'_q$ $\quad \rightarrow$ `Just` ($swaplabels\ i'_q$) (PreferJump2)
  | `pull _ _ _ _` $\leftarrow i'_q \rightarrow$ `Just` $i'_p$ $\quad\quad\quad\quad\quad$ (DeferPull1)
  | `pull _ _ _ _` $\leftarrow i'_p \rightarrow$ `Just` ($swaplabels\ i'_q$) (DeferPull2)
  (`Just` $i'_p$, `_`) $\quad\quad\quad\quad \rightarrow$ `Just` $i'_p$ $\quad\quad\quad\quad\quad$ (Run1)
  (`_`, `Just` $i'_q$) $\quad\quad\quad\quad \rightarrow$ `Just` ($swaplabels\ i'_q$) (Run2)
  (`Nothing`, `Nothing`)$\mapsto$ `Nothing` $\quad\quad\quad\quad\quad$ (Deadlock)

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

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 remain-

ing 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).

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.

$tryStep$ : $(Channel \mapsto ChannelType2) \rightarrow LabelF \rightarrow Instruction \rightarrow LabelF \rightarrow Maybe\ Instruction$
$tryStep\ cs\ (l_p, s_p)\ i_p\ (l_q, s_q)$ = match $i_p$ with

$\quad$ jump $(l', u')$ $\hfill$ (LocalJump)
$\qquad \rightarrow$ Just (jump $((l', s_p), (l_q, s_q), u')$)

$\quad$ case $e\ (l'_t, u'_t)\ (l'_f, u'_f)$ $\hfill$ (LocalCase)
$\qquad \rightarrow$ Just (case $e\ ((l'_t, s_p), (l_q, s_q), u'_t)\ ((l'_f, s_p), (l_q, s_q), u'_f)$)

$\quad$ push $c\ e\ (l', u')$
$\qquad |\ cs[c] =$ out1 $\hfill$ (LocalPush)
$\qquad \rightarrow$ Just (push $c\ e\ ((l', s_p), (l_q, s_q), u')$)
$\qquad |\ cs[c] =$ in1out1 $\wedge\ s_q[c] =$ none$_F$ $\hfill$ (SharedPush)
$\qquad \rightarrow$ Just (push $c\ e\ ((l', s_p), (l_q, s_q[c \mapsto$ pending$_F]), u'[$chan $c \mapsto e])$)

$\quad$ pull $c\ x\ (l'_o, u'_o)\ (l'_c, u'_c)$
$\qquad |\ cs[c] =$ in1 $\hfill$ (LocalPull)
$\qquad \rightarrow$ Just (pull $c\ x\ ((l'_o, s_p), (l_q, s_q), u'_o)\ ((l'_c, s_p), (l_q, s_q), u'_c)$)

$\qquad |\ (cs[c] =$ in2 $\vee\ cs[c] =$ in1out1$)\ \wedge\ s_p[c] =$ pending$_F$ $\hfill$ (SharedPullPending)
$\qquad \rightarrow$ Just (jump $((l'_o, s_p[c \mapsto$ have$_F]), (l_q, s_q), u'_o[x \mapsto$ chan $c])$)

$\qquad |\ (cs[c] =$ in2 $\vee\ cs[c] =$ in1out1$)\ \wedge\ s_p[c] =$ closed$_F$ $\hfill$ (SharedPullClosed)
$\qquad \rightarrow$ Just (jump $((l'_c, s_p), (l_q, s_q), u'_c)$)

$\qquad |\ cs[c] =$ in2 $\wedge\ s_p[c] =$ none$_F\ \wedge\ s_q[c] =$ none$_F$ $\hfill$ (SharedPullInject)
$\qquad \rightarrow$ Just (pull $c\ ($chan $c)$
$\qquad\qquad\qquad\quad\ ((l_p, s_p[c \mapsto$ pending$_F]), (l_q, s_q[c \mapsto$ pending$_F]), [])$
$\qquad\qquad\qquad\quad\ ((l_p, s_p[c \mapsto$ closed$_F]), (l_q, s_q[c \mapsto$ closed$_F]), []))$

$\quad$ drop $c\ (l', u')$
$\qquad |\ cs[c] =$ in1 $\hfill$ (LocalDrop)
$\qquad \rightarrow$ Just (drop $c\ ((l', s_p), (l_q, s_q), u')$)
$\qquad |\ cs[c] =$ in1out1 $\hfill$ (ConnectedDrop)
$\qquad \rightarrow$ Just (jump $((l', s_p[c \mapsto$ none$_F]), (l_q, s_q), u')$)
$\qquad |\ cs[c] =$ in2 $\wedge\ (s_q[c] =$ have$_F \vee s_q[c] =$ pending$_F)$ $\hfill$ (SharedDropOne)
$\qquad \rightarrow$ Just (jump $((l', s_p[c \mapsto$ none$_F]), (l_q, s_q), u')$)
$\qquad |\ cs[c] =$ in2 $\wedge\ s_q[c] =$ none$_F$ $\hfill$ (SharedDropBoth)
$\qquad \rightarrow$ Just (drop $c\ ((l', s_p[c \mapsto$ none$_F]), (l_q, s_q), u')$)

$\quad$ close $c\ (l', u')$
$\qquad |\ cs[c] =$ out1 $\hfill$ (LocalClose)
$\qquad \rightarrow$ Just (close $c\ ((l', s_p), (l_q, s_q), u')$)
$\qquad |\ cs[c] =$ in1out1 $\wedge\ s_q[c] =$ none$_F$ $\hfill$ (SharedClose)
$\qquad \rightarrow$ Just (close $c\ ((l', s_p), (l_q, s_q[c \mapsto$ closed$_F]), u')$)

$\quad$ exit $\hfill$ (LocalExit)
$\qquad \rightarrow$ Just exit

$\quad$ _ | otherwise $\hfill$ (Blocked)
$\qquad \rightarrow$ 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 $\mathsf{none}_F$. When this constraint is satisfied, the result instruction stores the pushed value in the stream buffer variable (chan $c$) and sets the static input state to $\mathsf{pending}_F$, 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 $\mathsf{pending}_F$. The result instruction copies the value from the stream buffer variable into a variable specific to the left source process. The corresponding $\mathsf{have}_F$ 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.

$$
\begin{aligned}
\textit{channels} \quad &: \quad \textit{Process} \rightarrow \textit{Process} \rightarrow (\textit{Channel} \mapsto \textit{ChannelType}_2) \\
\textit{channels } p \; q \quad &= \quad \{c = \mathtt{in2} \quad | \; c \in (\mathtt{ins}\; p \cap \mathtt{ins}\; q)\} \\
&\quad \cup \; \{c = \mathtt{in1} \quad | \; c \in (\mathtt{ins}\; p \cup \mathtt{ins}\; q) \land c \notin (\mathtt{outs}\; p \cup \mathtt{outs}\; q)\} \\
&\quad \cup \; \{c = \mathtt{in1out1} \;| \; c \in (\mathtt{ins}\; p \cup \mathtt{ins}\; q) \land c \in (\mathtt{outs}\; p \cup \mathtt{outs}\; q)\} \\
&\quad \cup \; \{c = \mathtt{out1} \quad | \; c \notin (\mathtt{ins}\; p \cup \mathtt{ins}\; q) \land c \in (\mathtt{outs}\; p \cup \mathtt{outs}\; q)\}
\end{aligned}
$$

$$
\begin{aligned}
\textit{outlabels} \quad &: \quad \textit{Instruction} \rightarrow \{\textit{Label}\} \\
\textit{outlabels}\,(\mathtt{pull}\; c \; x \; (l, u) \; (l', u')) \quad &= \quad \{l, l'\} \\
\textit{outlabels}\,(\mathtt{drop}\; c \; (l, u)) \quad &= \quad \{l\} \\
\textit{outlabels}\,(\mathtt{push}\; c \; e \; (l, u)) \quad &= \quad \{l\} \\
\textit{outlabels}\,(\mathtt{close}\; c \; (l, u)) \quad &= \quad \{l\} \\
\textit{outlabels}\,(\mathtt{case}\; e \; (l, u) \; (l', u')) \quad &= \quad \{l, l'\} \\
\textit{outlabels}\,(\mathtt{jump}\; (l, u)) \quad &= \quad \{l\} \\
\textit{outlabels}\,(\mathtt{exit}) \quad &= \quad \{\}
\end{aligned}
$$

$$
\begin{aligned}
\textit{swaplabels} \quad &: \quad \textit{Instruction} \rightarrow \textit{Instruction} \\
\textit{swaplabels}\,(\mathtt{pull}\; c \; x \; ((l_1, l_2), u) \; ((l_1', l_2'), u')) &= \mathtt{pull}\; c \; x \; ((l_2, l_1), u) \; ((l_2', l_1'), u') \\
\textit{swaplabels}\,(\mathtt{drop}\; c \; ((l_1, l_2), u)) &= \mathtt{drop}\; c \; ((l_2, l_1), u) \\
\textit{swaplabels}\,(\mathtt{push}\; c \; e \; ((l_1, l_2), u)) &= \mathtt{push}\; c \; e \; ((l_2, l_1), u) \\
\textit{swaplabels}\,(\mathtt{close}\; c \; ((l_1, l_2), u)) &= \mathtt{close}\; c \; ((l_2, l_1), u) \\
\textit{swaplabels}\,(\mathtt{case}\; e \; ((l_1, l_2), u) \; ((l_1', l_2'), u')) &= \mathtt{case}\; e \; ((l_2, l_1), u) \; ((l_2', l_1'), u') \\
\textit{swaplabels}\,(\mathtt{jump}\; ((l_1, l_2), u)) &= \mathtt{jump}\; ((l_2, l_1), u) \\
\textit{swaplabels}\,(\mathtt{exit}) &= \mathtt{exit}
\end{aligned}
$$

Figure 4.14: Utility functions for fusion

```
                    stock

                      |
                      |
                      v

                     map
                    /    \
                   /      \
                  v        v

        correlation       regression
```
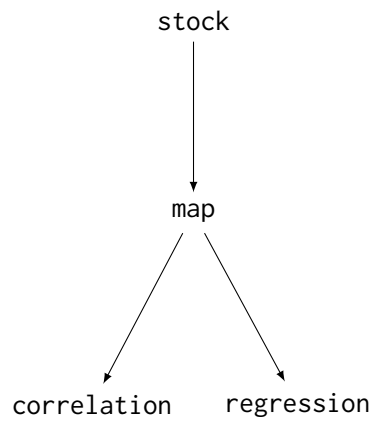
Figure 4.15: Dependency graph for `priceOverTime` example

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.

## 4.4 SYNCHRONISING PULLING BY DROPPING

The `drop` instruction exists to synchronise two consumers of the same input, so that both processes pull the same value at roughly the same time. When fusing two consumers, the fusion algorithm uses drops when coordinating the processes 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

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

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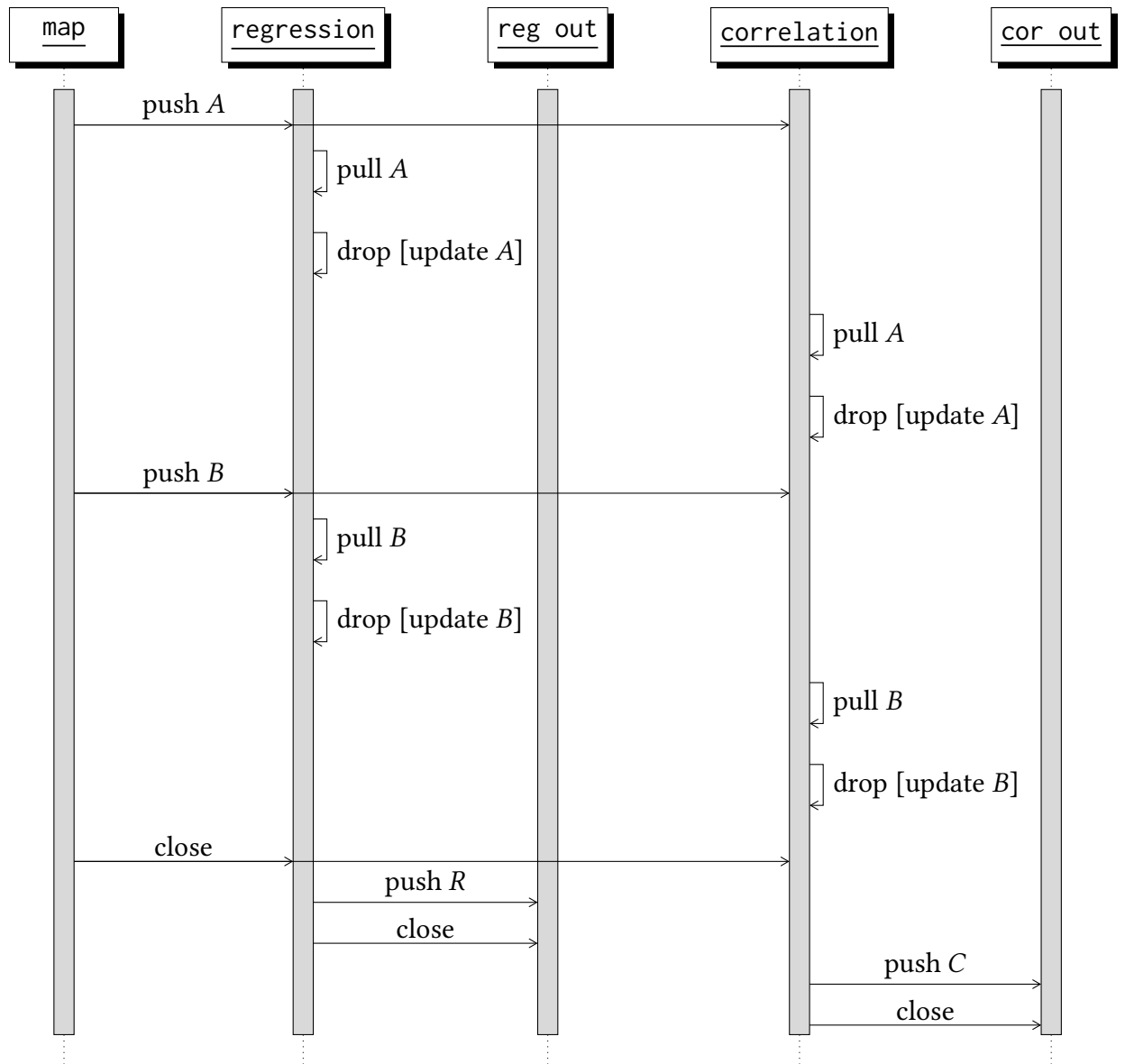sation between `regression` and `correlation` processes, we use placeholder values such as *A* and *B* instead of actual stream values, and show only a subset of the whole execution, omitting the `stock` input stream and the internal messages of the `map` process.

In the definition of the `fold` process template, the `drop` instruction also 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 `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`. 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 4.17 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
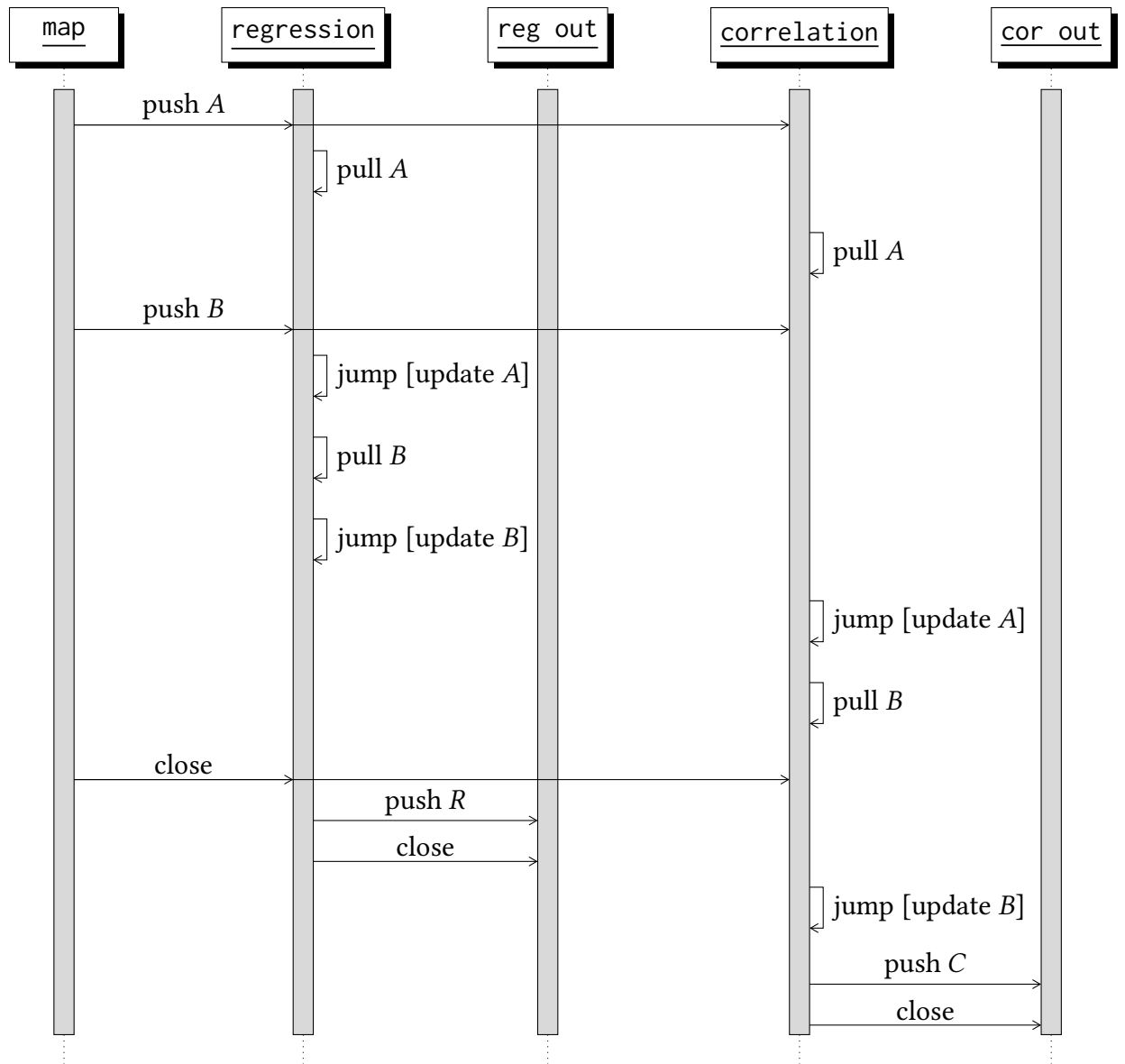
Figure 4.17: Sequence diagram for a possible linearised execution of `priceOverTime`, using a hypothetical semantics without drop synchronisation

*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 input processes. For process networks which contain more than two processes, we repeatedly fuse pairs of processes in the network together until only one process remains.

When fusing a pair of processes, the fused process tends to have more states than each input process individually, because the fused process has to do the work of both input processes. In general, the larger the input processes, the larger the fused process will be, and when we have many processes to fuse, the result will get progressively larger as we fuse more processes in. If the fused 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 re-
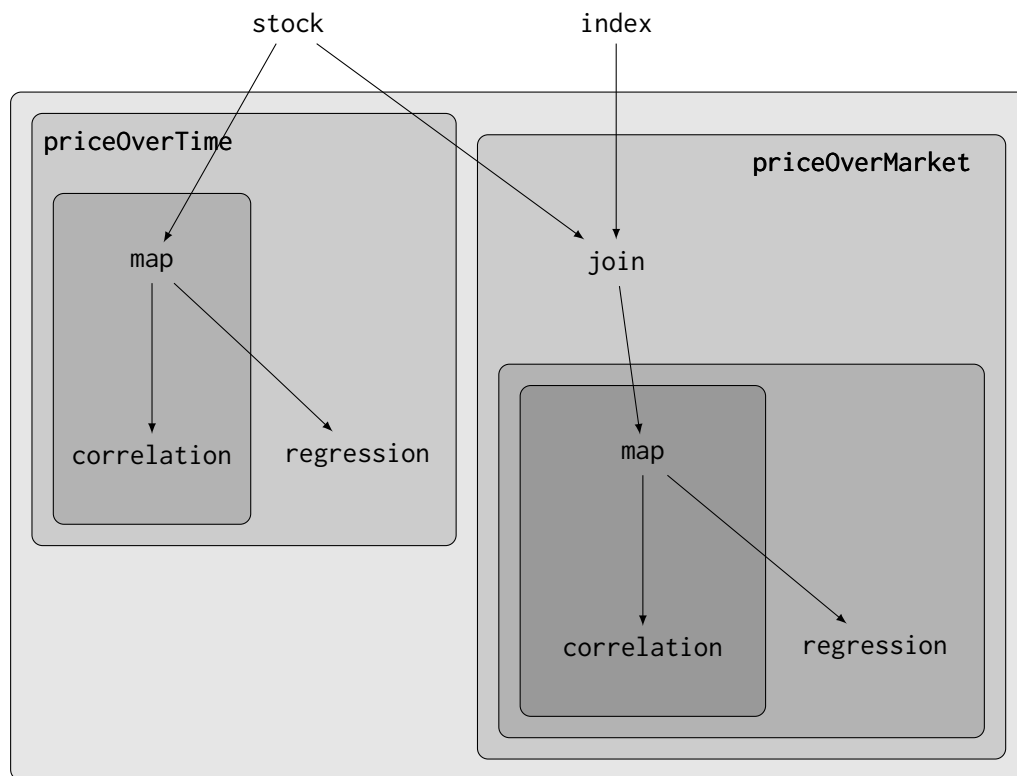
Figure 4.18: Pairwise fusion ordering of the `priceAnalyses` network.

peatedly fusing the pairs of processes in a network, we perform some simplifications between each fusion step, to remove unnecessary states and simplify the input for 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 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, and the more deeply nested
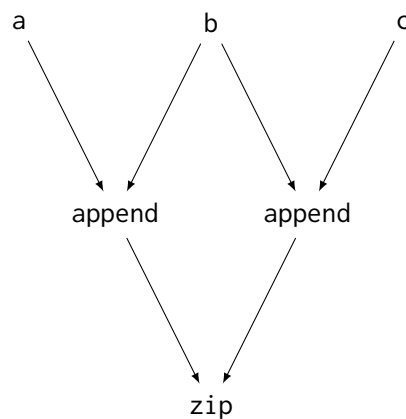
Figure 4.19: Process network for append2zip

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. Each input stream pushes its elements to its consumers; 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 when it pushes elements, it pushes to both its 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.
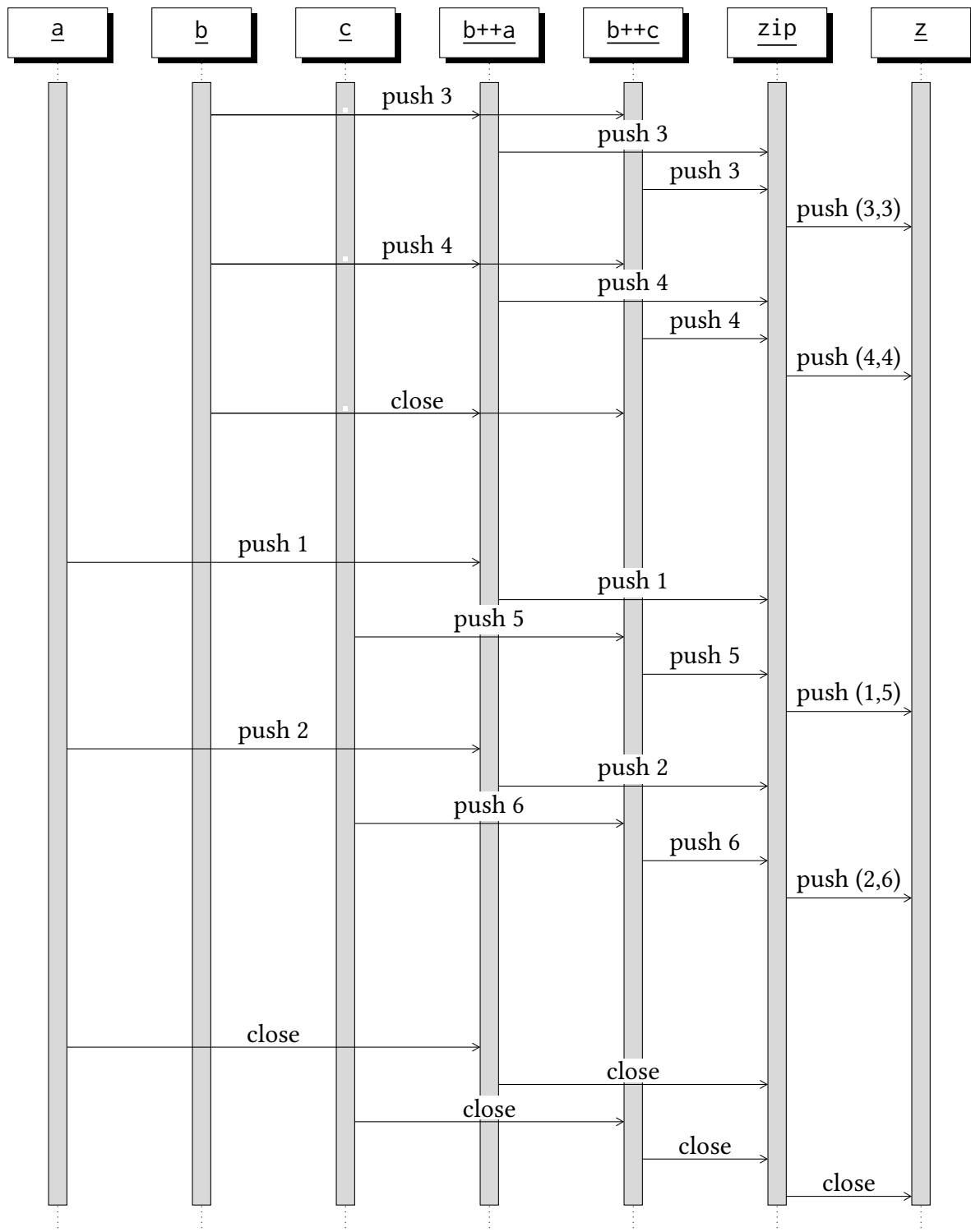
Figure 4.20: Sequence diagram of execution of append2zip

This example contains three processes. 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 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 sink. 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 err on the side of caution, telling the programmer about anything even slightly unexpected. Rather than making the decision of which order to execute them in, we display a compile-time error and make the programmer separate the 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 pessimistic approximation. In the implementation, if at any point we encounter a pair of processes which we cannot fuse together, we display a compile-time error telling the programmer that the network cannot be fused.

Unfortunately, this 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])
```
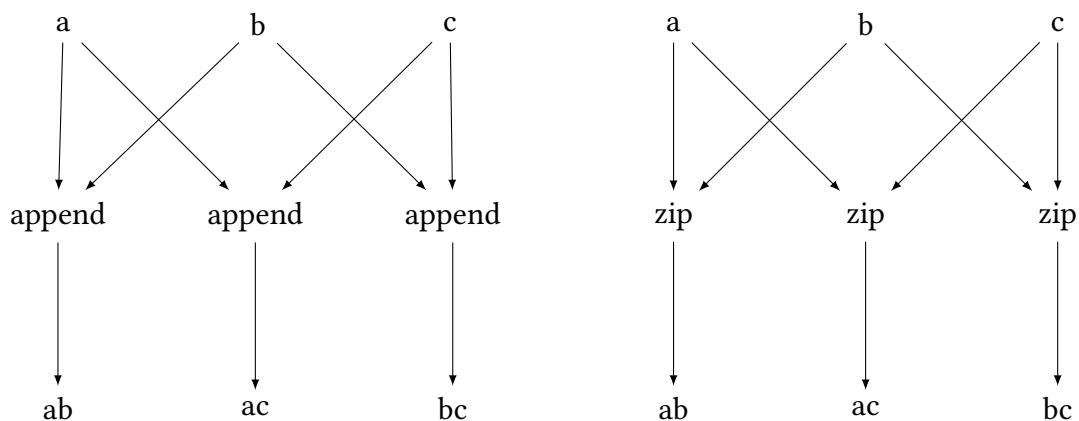
Figure 4.21: Process networks for `append3` and `zip3`

```
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.

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 7.1.

## 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 the two source processes may also produce that same sequence. Note that due to the non-determinism of process execution, the converse is not true for all input 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. This proof 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.
```

---

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

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).

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 a precondition, which is expressed as a predicate of the evaluation state. The precondition for the initial label must be true for the initial evaluation state. Whenever the program takes an evaluation step, assuming the precondition for the original

label was satisfied at the start of the step, the precondition for the result label must be satisfied by the updated evaluation state. With these two conditions satisfied, we can show that all evaluations respect the preconditions by performing induction over the evaluation relation.

The proof itself is not novel, except for the fact that it is mechanised. We believe that the proof gives sufficient confidence in the correctness of the presentation given earlier, despite the differences in formulation.

# CHAPTER 5

# EVALUATION

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 fusion 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

To benchmark our algorithm, we have implemented the fusion algorithm described in chapter 4, in a library called *Folderol*[1]. This fusion system uses the topology of the entire process network to perform fusion. When fusing one producer with multiple consumers, the fusion algorithm must coordinate between all the consumers to reach consensus on when to pull the next value. This coordination between all consumers means the fusion algorithm requires global knowledge of the process network. In contrast, shortcut fusion systems such as (Gill et al., 1993) use rewrite rules to remove intermediate buffers and require only local knowledge, 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: if we wish to know whether all the processes have been fused,

---

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

we need to know about all the processes. To fuse the entire process network at compile-time, as well as to inform the programmer when fusion fails, Folderol uses Template Haskell, a form of metaprogramming.

Benchmarks are available at `https://github.com/amosr/folderol/tree/master/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'(Thompson, 2015).

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 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. 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 are

pull-based, and 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. If the function that computes the next step of the stream is hidden inside a monadic computation, it is hard for the compiler to pull it out and construct it. Because the structure — the function that computes the rest of the stream — is dynamic, it is hard for the compiler to reason statically about the structure. In Vector's stream fusion, great effort is taken to make the step function non-recursive and statically known, by splitting the stream into a static step function and a dynamic state. Vector's static step function can be optimised at compile-time, while the state can change dynamically. In contrast, in Conduit and Pipes, the step function and state are mixed together: the remainder of the stream is only available dynamically as a function closure, hidden inside the continuation for a monadic operation.

The final streaming library, 'Streaming', is similar to the previous two libraries, in that it is a monadic streaming library with limited static fusion. Unlike the previous two, Streaming supports explicit duplication of streams, in a similar way to polarised streams (section 2.4). To duplicate the stream, the `store` function takes a computation to perform on the copy of the stream, while the original stream is passed through. 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 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. To run the benchmarks, we used the Criterion[2] library. Criterion offers a fairly reliable way to evaluate

---

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

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 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 5.1 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 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

```
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 5.1: Folderol implementation of `priceAnalyses`

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

Figure 5.1: Runtime performance for `priceAnalyses` queries

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 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 is 1.5 times faster than the Pipes implementation, and 3.7 times slower than the fused version.

```
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 5.2: Folderol implementation of `filterMax`

### 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. The streaming libraries, including Folderol, are parameterised over the monad for effects. For simplicity we use an `IO` monad.

Listing 5.2 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

```
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 5.3: Vector / share implementation of filterMax

annot stream is filtered to only those above the line (above), then the annotations thrown away
(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 5.3, the corresponding annot vector cannot be fused with both of its con-
sumers, 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 manifesting the distances vector and sharing it among the two consumer loops,
we can also recompute the distances for each consumer. The recomputed distances implemen-
tation 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.

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 allo-
cation. 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 A.7), while the second is hand-fused into a single pass (listing A.8). 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.9), which follows much the same structure as the Conduit hand-fused version.

Finally, the Streaming version (listing A.10) 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 Vector versions are somewhat slower because they require two iterations over the data, but 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 signif-

|  |  | 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

icantly slower, spending 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.
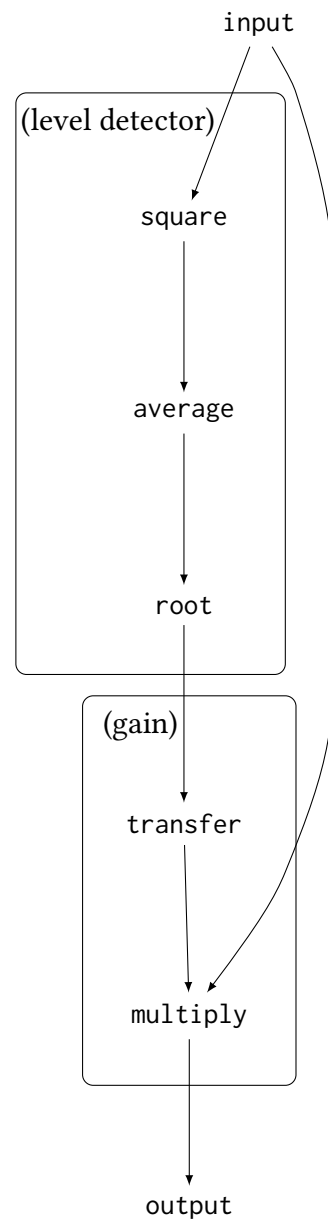
Figure 5.2: Dependency graph for audio compressor

### 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 5.4. 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.11) 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.

```
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 5.4: 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 5.5: Folderol implementation of compressor with low-pass

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 are 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.

The Folderol implementation of a dynamic range compressor with a low-pass is shown in listing 5.5. The low-pass filter is itself a kind of moving average, and is defined in much the

|         | 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

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.12), 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.

### 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 5.6 shows the Folderol implementation. In Conduit (listing A.13) and Pipes (listing A.14), 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.

```
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 5.6: 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

```
part2 :: FilePath → FilePath → FilePath → IO (Int, Int)
part2 fpIn1 fpOut1 fpOut2 = do
  (c1,(c2,())) ← scalarIO $ λsnkC1 → scalarIO $ λsnkC2 →
    $$(fuse defaultFuseOptions $ 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 5.7: 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

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 5.7 shows the Folderol implementation. Even with partial hand-fusion because of the multiple outputs, the Pipes (listing A.18) and Conduit (listing A.17) programs are slower than ours, as well as losing the abstraction benefits from using a high-level library. The Streaming implementation (listing A.20) allows streams to be shared in a fairly straightforward way and does not require hand-fusion, but is also the slowest in this benchmark. Table 5.5 shows the runtimes for partitioning a file with one million lines, around 6.5MB.

### 5.1.6 *Partition / append*

The program in listing 5.8 partitions an input vector into a vector containing the even-valued elements, and a vector containing the odd-valued elements. The two result vectors are then appended together. If we try to compile this code with Folderol, we get a compile-time warning:

```
bench/Bench/PartitionAppend/Folderol.hs:18:8: warning:
  Maximum process count exceeded:
    after fusion there are 2 processes, but maximum is 1.
  Falling back to unbounded communication channels.
```

```
partitionAppendFailure :: Vector Int → IO (Vector Int)
partitionAppendFailure xs = do
  (apps,()) ← vectorSize xs $ λsnkApps →
    $$(fuse $ do
        x0    ← source [|sourceOfVector xs|]
        evens ← filter [|even            |] x0
        odds  ← filter [|odd             |] x0
        — Failure: cannot fuse append with both filters
        apps  ← append evens odds
        sink apps       [|snkApps         |]$$)
  return apps
```

Listing 5.8: Partition / append fusion failure

This warning indicates that the program could not be fused into a single loop without introducing unbounded buffering. Some sort of buffering is inevitable, 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, 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 this case, the input stream is backed by a real vector, which means we can, in fact, implement it without introducing an unbounded buffer: we already have the buffer. To execute the program, we explicitly introduce another source to read from the vector, as in listing 5.9. The two filters now apply to different input sources, and 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 5.10.

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

Listing 5.9: Partition / append with two sources

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

Listing 5.10: Partition / append with two loops

|        |          | Runtime (s) | Runtime (%) | Allocation (b) | Allocation (%) |
|--------|----------|------------:|------------:|---------------:|---------------:|
| Folderol | 2-loop   | 0.17s       | 100%        | 2.4e8          | 100%           |
|        | 2-source | 0.28s       | 165%        | 8.0e7          | 33%            |
| Vector | 2-loop   | 0.15s       | 88%         | 1.6e8          | 67%            |
|        | 2-source | 0.26s       | 153%        | 1.6e8          | 67%            |

Table 5.6: PartitionAppend2 benchmark results

The Vector implementations (listing A.21) follow the same structure: 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 it 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 back-to-front, can be reversed in-place. Unfortunately, in this example, the Vector program still splits this array in two before appending it 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 - a third of the two-loop Folderol, and a half of the Vector versions. 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 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 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
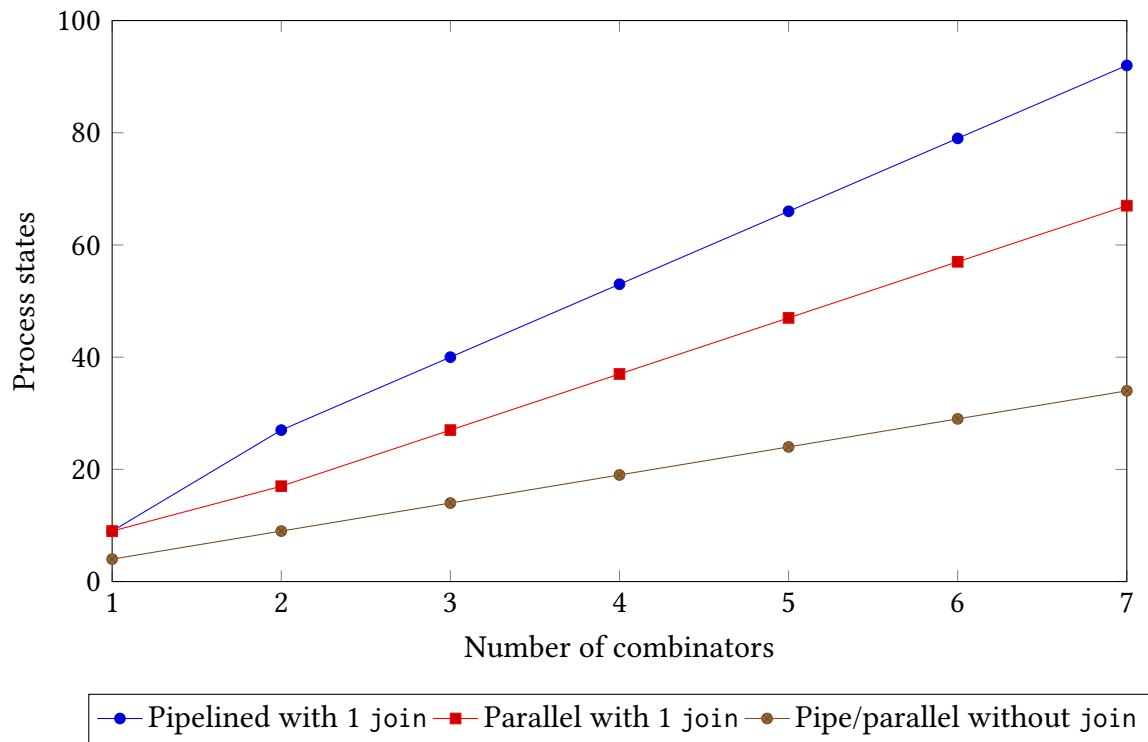
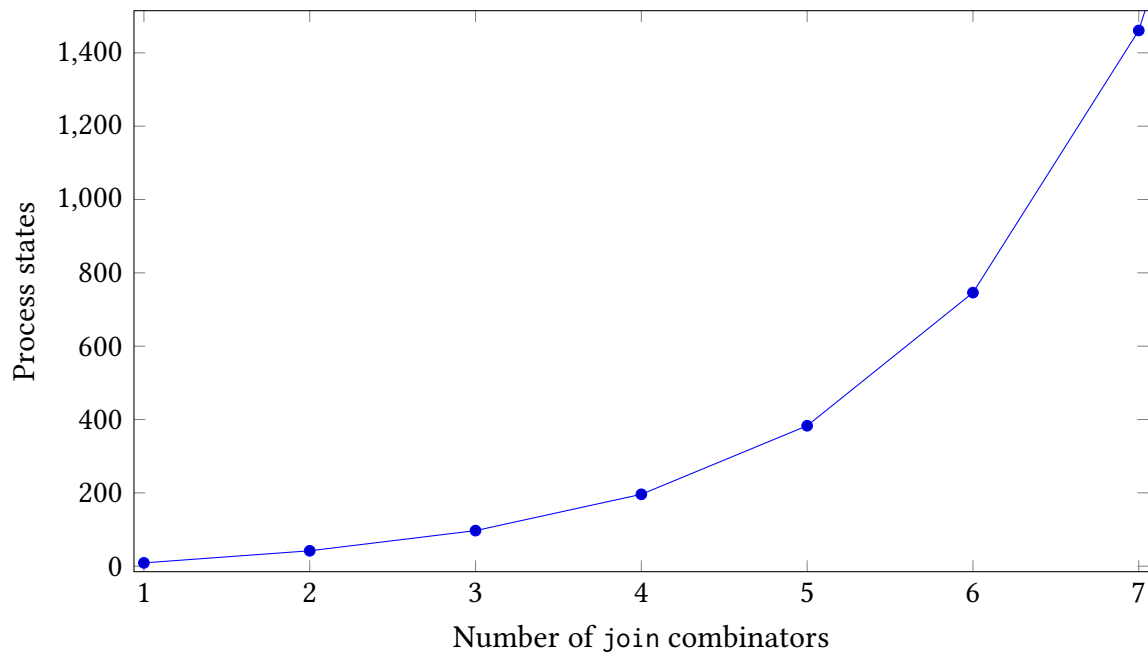Figure 5.3: Maximum output process size for fusing all combinations of up to $n$ combinators



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

`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.

## 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 read through Core programs to know whether whether fusion has worked or failed. If fusion succeeds, we know it has succeeded. If fusion fails, we know not just that it has failed, but *where* it has failed. These explicit fusion failures make it a lot easier to track down performance issues due to non-fusion.

CHAPTER 6

RELATED WORK

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

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 or push-based streams are sufficient to execute multiple queries. Some combinators are inherently push-based, particularly those with multiple outputs such as unzip; while others are inherently pull-based, such as zip.

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 input to be *linear* in the input data structures. 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. However, shortcut fusion relies on local inlining which, like pull-based streams, only occurs when there is a single consumer. Thus, shortcut fusion is inherently biased towards

pull fusion. Push-based shortcut fusion systems *do* exist (Gill et al., 1993), but support neither `zip` nor `unzip` (Svenningsson, 2002; Lippmeier et al., 2013).

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. This allows unfusable programs to be caught at compile time. However, it 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. However, while these libraries provide boundedness guarantees, they provide no fusion guarantees, and as such programs tend to be written over chunks of data to make up for the communication

overhead. For the most part they support only straight-line computations with only limited forms of branching, while Streaming supports explicit duplication in a similar way to polarised streams. We compared the runtime performance of these earlier, in chapter 5.

The benefits of fusion have been known about for a long time, since at least the 1970s. Jackson Structured Programming (Jackson, 2002) is a design methodology where the structure of a program is derived using the structure of the input files to process. Here, a method called "program inversion" 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   SYNCHRONISED PRODUCT

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. This is a much simpler fusion method than ours, 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: they do not have to take shared steps at the same time, but if one process lags behind the other, it must catch up before the other one 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 on this have been limited.

## 6.3    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. This is necessary for combinators such as mergesort-style `merge`, as well as `append`. Note that in the synchronous language literature, it is common to refer to a different merge operation, also known as `default`, which computes a stream that is defined whenever either input is defined.

## 6.4    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 merge 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 still 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. This does appear to be able to express value-dependent operations such as merge, but lacks the composability — and familiarity — of combinators.

## 6.5   TUPLING

Automatic tupling combines multiple traversals over a data structure into a single traversal. Tupling is more general than stream fusion: it supports simplifying 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 strucutre, both traversals can be computed together. Automatic 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.

## 6.6    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 was an unfair comparison between a compiled Neumann push model and an un-optimised pull model (Shaikhha et al., 2018).

This control-flow is the same as for *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); neither does the Neumann push model support combinators with multiple inputs except append; nor does it support executing multiple queries concurrently.

Biboudis (2017) describes how this model is better for Java JIT compilation, 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.

# CONCLUSION

This final chapter discusses some directions for future work, before concluding the thesis. This thesis has presented four 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 grpah.

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 fusion transform.

Finally, 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 slightly faster than the compared array fusion implementation.

If, in the future of computing, interest in *big data* does not wither and fade; if datasets continue to grow faster than hard drives or memory; if data analysis is required to answer questions; then, the concurrent execution of streaming queries will only become more important.

## 7.1    FUTURE WORK

In this last section, we take a brief look at the limitations and possible extensions of our work.

### 7.1.1    *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 are moving the non-determinism from the fusion algorithm deciding which process to execute, and reifying it in the result process itself.

Reifying the non-determinism in the processes will mean that all fusion orders produce the same process at the end. Different orders will not affect the result, or whether things fuse. Different orders do affect the size of the intermediate process, after performing some fusion but before all processes are fused together. Fusing two unconnected processes which read from different streams introduces a lot of non-determinism: at each step of the fused process, either of the original processes can take a step. The two processes do not constrain each other, and the result process will have a lot of states. Fusing connected processes, for example a producer and a consumer, introduce less non-determinism because there are points when only one of the processes can run. When the consumer is waiting for a value, only the producer can run. 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 not any different, 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.

### 7.1.2  *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 sim-

plification 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.

# BIBLIOGRAPHY

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*.

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.

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.

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.*

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*.

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.

Gonzalez, G. (2012). The pipes Haskell package. `http://hackage.haskell.org/package/pipes`.

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.

Johnston, W. M., Hanna, J., and Millar, R. J. (2004). Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*.

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*.

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*.

McSherry, F., Isard, M., and Murray, D. G. (2015). Scalability! But at what COST? In *Hot Topics in Operating Systems*.

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.

Parks, T. M. (1995). *Bounded scheduling of process networks*. PhD thesis, University of California. Berkeley, California.

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.

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`.

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*.

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.

# CHAPTER A

## BENCHMARK CODE

```
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

```
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 A.7: Conduit two-pass implementation of filterMax

```
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 A.8: Conduit one-pass (hand-fused) 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.9: Pipes 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 A.10: Streaming implementation of `filterMax`

```haskell
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.11: Vector implementation of compressor

```haskell
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.12: Vector implementation of compressor with low-pass

```haskell
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.13: 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.14: 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.15: 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.16: Streaming implementation of append2

```
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 A.17: Conduit implementation of part2

```
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.18: 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.19: Hand 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 A.20: Streaming implementation of part2

```haskell
partitionAppendV2Loop :: Unbox.Vector Int → IO (Unbox.Vector Int)
partitionAppendV2Loop !xs = do
  let (as,bs) = Unbox.partition (λi → i `mod` 2 ≡ 0) xs
  let  asbs   = as Unbox.++ bs
  return asbs

partitionAppendV2Source :: Unbox.Vector Int → IO (Unbox.Vector Int)
partitionAppendV2Source !xs = do
  let p i     = i `mod` 2 ≡ 0
  let as      = Unbox.filter          p  xs
  let bs      = Unbox.filter (not ∘ p) xs
  let asbs    = as Unbox.++ bs
  return asbs
```

Listing A.21: Vector implementations of partitionAppend