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, **??**).

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 **??** we introduce a query language called Icicle, which allows programmers to write and reason about queries using a more familiar array-based semantics, while retaining the execution strategy of push streams. The type system of Icicle guarantees that well-typed query programs have the same semantics whether they are executed as array programs or as stream programs, and that all queries over the same input can be executed together in a single pass.

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 **??** we introduce *process fusion*, a method for taking multiple processes in a Kahn process network and fusing them together into a single process. The fused process communicates through local variables rather than costly communication channels. 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.

Streaming programs are restricted to a single pass over the input, but if the input data is persistent, for example a file on disk or an array in memory, we can perform multiple passes by executing each pass separately as its own streaming operation. We call such streams *rewindable*, as they can be "rewinded" back to the start after reading. For programs requiring multiple passes over the input, there are generally many different ways to divide the work among the passes. We perform *clustering* on the program to determine how many passes to perform, and how to schedule each stream operation among the different passes. The choice of clustering affects runtime performance. To minimise the time spent reading and re-reading the data, we would like to use a clustering which requires the least number of passes. Finding this clustering is NP-hard (Darte, 1999).

In chapter 3, we find the clustering by generating an Integer Linear Program, which can be solved by an external solver. The clustering algorithm of Megiddo and Sarkar (1997) computes the clustering for an imperative loop nest, rather than a set of stream transformers. Individual loops in the loop nest are assigned to clusters, and all loops in the same cluster are fused together. Here, as with many imperative loop fusion systems, loops can only be fused together if their loop bounds are identical. In such systems, a loop that filters an array cannot be fused with a loop that consumes the filtered array, as they have different loop bounds. With process fusion, we *can* fuse a filter with its consumer, so a clustering algorithm for imperative loop nests would introduce more passes than necessary. Our clustering algorithm is an extension Megiddo and Sarkar (1997) to cluster stream transformers, and to support size-changing operations such as filter. [todo *refine clustering*]

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 **??**.

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 **??**.

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 **??**.

CLUSTERING FOR REWINDABLE STREAMS: a clustering algorithm for streaming transformers, which supports size-changing operations such as filter. This algorithm converts a set of streaming transformers to an Integer Linear Program to be solved externally. The clustering algorithm is described in chapter 3. [todo *refine*]

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 **??**, 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:

```
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.

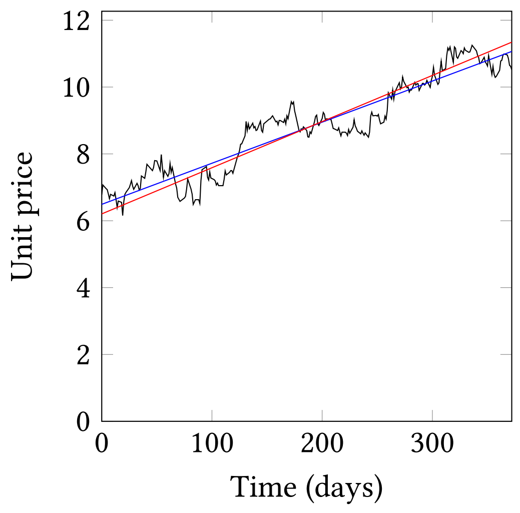


Figure 2.1: Analysis of a stock over a year

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

```
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)
```

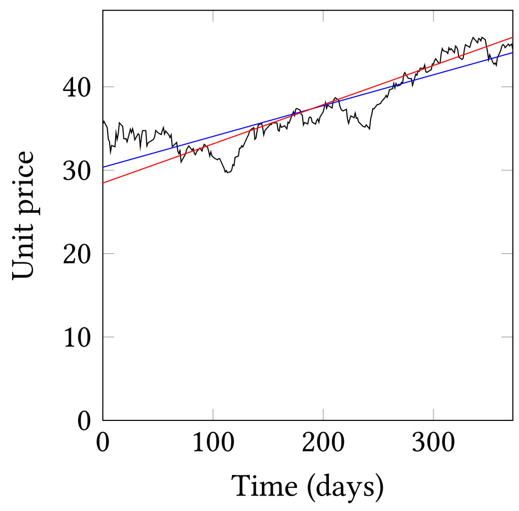
Both `regression` and `correlation` functions take a list of pairs of numbers, so we first convert the `Record` values to pairs of numbers using `map`. Although this is a single program, it computes two values. Whether we think of this program as one query or two is inconsequential; the important part 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.

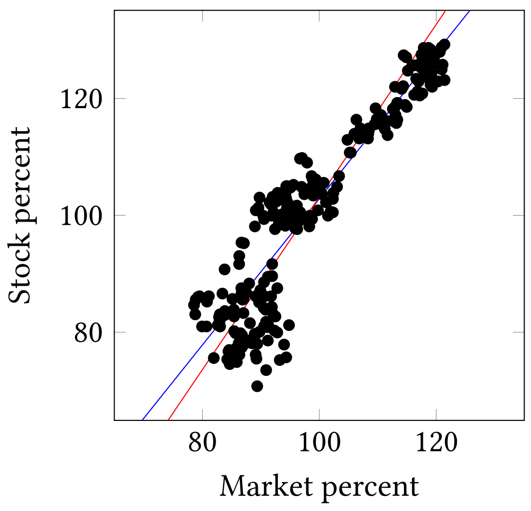
To compare against the rest of the market, we have another file of records containing the average price of a representative subset of stocks. This representative subset is called a *market index*. We want to compare each day's price for our stock against the average price for the corresponding day in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

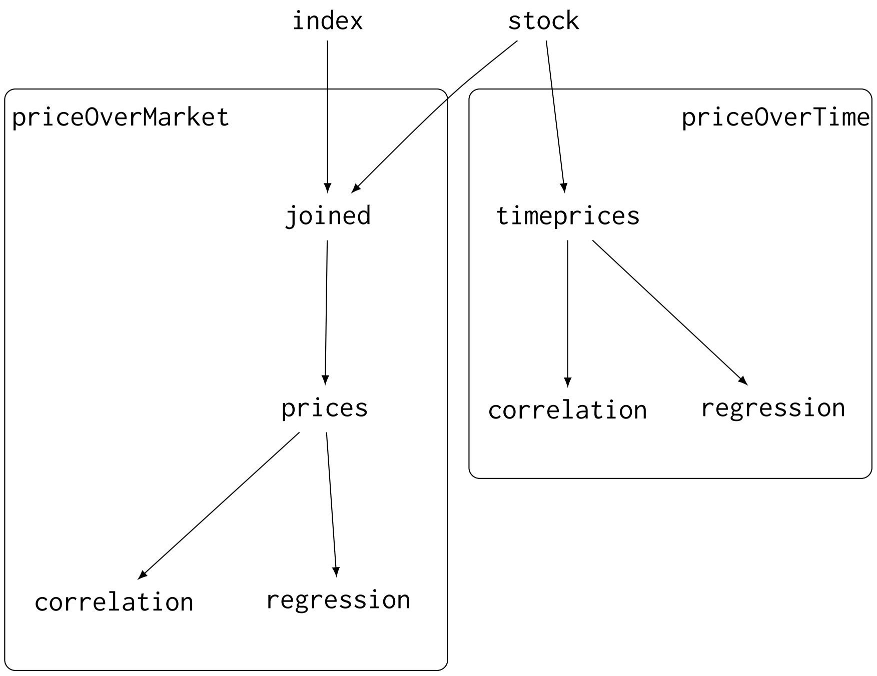


Figure 2.3: Dependency graph for queries `priceOverTime` and `priceOverMarket`

```
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.

```
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 ??.

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 **??** 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 **??**.

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 **??**.

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

```
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

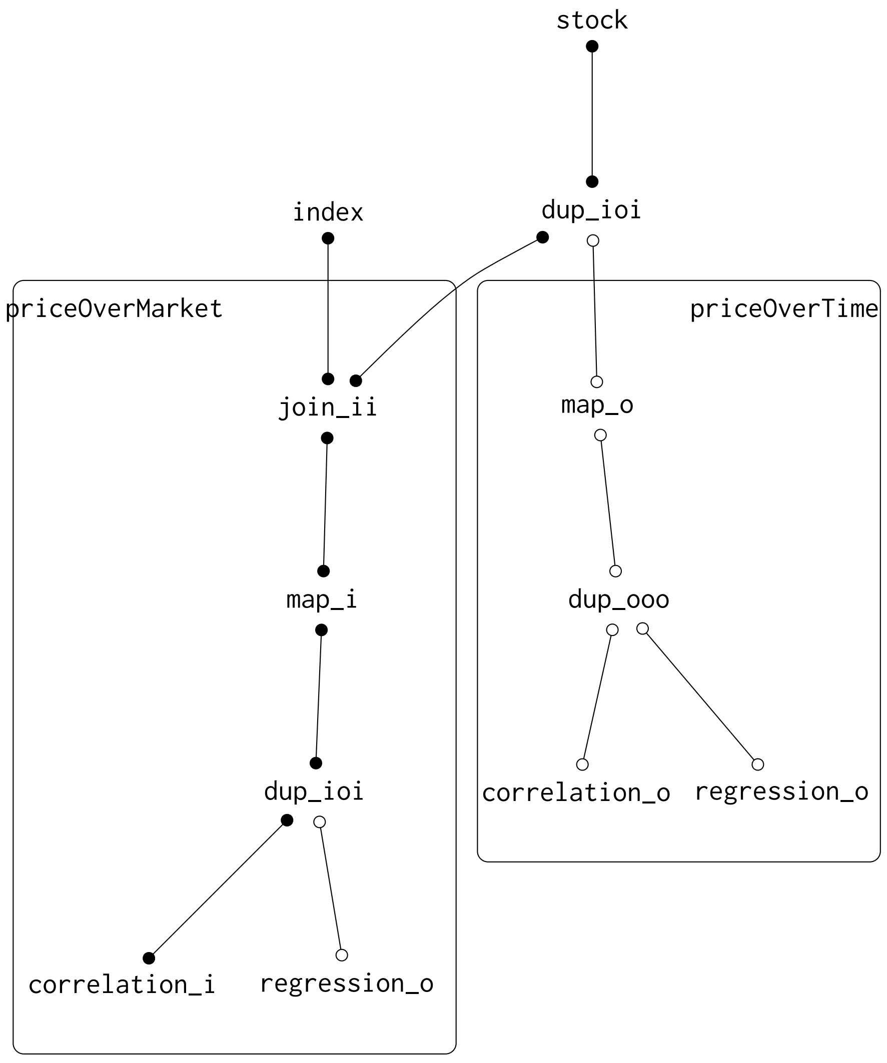


Figure 2.4: Polarised dependency graph for `priceOverTime` and `priceOverMarket`

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`

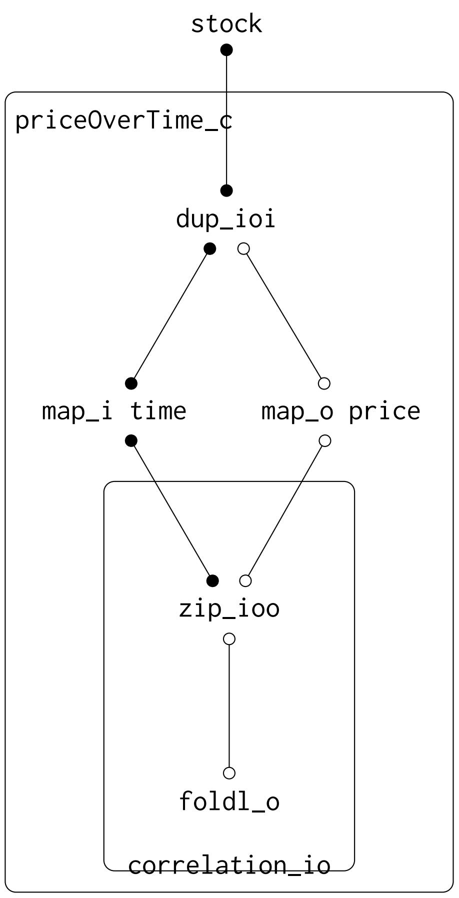


Figure 2.5: Polarised dependency graph with diamond

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

```
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 **??**, 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 **??** 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 **??**, 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 (**??**). 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 3

CLUSTERING FOR REWINDABLE STREAMS

This chapter presents a clustering algorithm for scheduling streaming programs that require multiple passes over input streams. This work was first published as Robinson et al. (2014). So far, in this thesis, we have assumed that all input streams are *ephemeral*: that is, once the elements have been read, they cannot be recovered. This property is certainly the case for reading from a network socket, but To support rewindable streams — such as reading from a disk — as well as ephemeral streams — such as reading from a network socket — the definition of process networks in **??** restricted the streaming programs to a single pass over the input. When we know that all the input streams are rewindable, we can execute larger programs which require multiple passes over the input, by executing each pass as its own streaming program. Given such a multiple pass program, we perform *clustering* to determine how many passes to perform, and how to schedule each stream operation among the different passes. The choice of clustering affects runtime performance. To minimise the time spent reading and re-reading the data, we would like to use a clustering which requires the least number of passes. Finding this clustering is NP-hard (Darte, 1999).

The contributions of this chapter are:

- We extend the clustering algorithm of Megiddo and Sarkar (1997) and Darte and Huard (2002), with support for size-changing operators. Size-changing operators can be clustered with operations on both their source stream and output stream, and compiled naturally with process fusion (section 3.4);

- We present a simplification to constraint generation that is also applicable to some ILP formulations such as Megiddo's: constraints between two nodes need not be generated if there is a fusion-preventing path between the two (section 3.4.5);

- Our constraint system encodes the cost model as a total ordering on the cost of clusterings, expressed using weights on the integer linear program. For example, we encode that memory traffic is more expensive than loop overheads, so given a choice between the two, memory traffic will be reduced (section 3.4.4);

- We present benchmarks of our algorithm applied to several common programming patterns. Our algorithm is complete and the chosen cost model yields good results in practice, though an optimal cost model is infeasible in general (section 3.5).

An implementation of our clustering algorithm is available at `https://github.com/amosr/clustering`.

3.1 INTRODUCTION

A collection-oriented programming model is ideal for expressing data-parallel computations, as it exposes the communication patterns to the compiler without requiring complicated loop analysis. It is essential, however, that the compiler combines these operations into a small number of efficient loops, as a naive translation leads to high memory traffic and poor performance. This is a well known problem, and techniques to fuse subsequent array operations together have been presented in (Gill et al., 1993; Coutts et al., 2007; Keller et al., 2010), to name just a few. This type of fusion alone is not generally sufficient to minimise memory traffic. As shown in Megiddo (Megiddo and Sarkar, 1997) and Darte (Darte and Huard, 2002), Integer Linear Programming can be used to find good clusterings. Unfortunately, they cannot handle operations like filter, where the output size differs from the input size. We present a technique that can handle both multi-loop fragments as well as size-altering operations.

To compile the clusters found by our clustering technique into sequential loops, we use data flow fusion (Lippmeier et al., 2013). It improved on existing array fusion approaches (Coutts et al., 2007; Keller et al., 2010) as it guarantees fusion into a single loop for programs that operate on the same size input data and contain no fusion-preventing dependencies between operators.

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

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

The function `normalize2` computes two sums, one of all the elements of `xs`, the other of only elements greater than zero. Since we need to fully evalute the sums to proceed with the maps, it is clear that we need at least two separate loops. These folds are examples of fusion-preventing dependencies, as fold cannot be fused with subsequent operations. Figure 3.1 shows the data-flow graph of `normalize2`. In the leftmost diagram we can see the effect of applying stream fusion to the program: we end up with four loops (denoted by the dotted lines): only the filter operation is combined with the subsequent fold. The best existing ILP approach results in the rightmost graph: it combines the `sum1` fold and `sum2` filter in one loop, but requires an extra loop for the fold operation which consumes the filter output, since it cannot fuse filter operations. Our approach, in the middle, produces the optimal solution in this case: one loop for the sums, another for the maps.

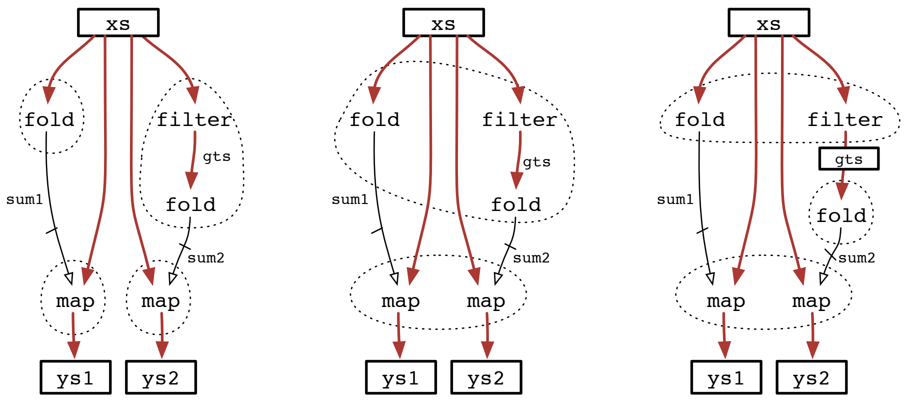


Figure 3.1: Clusterings for normalize2 example: with stream fusion; our system; best imperative system

3.2 COMBINATOR NORMAL FORM

Input programs are expressed in *Combinator Normal Form* (CNF), which is a textual description of the data flow graph. The grammar for CNF is given in figure 3.2. The `normalize2` example on the previous page is in CNF, as is the matching data flow graph for `normalize2` in figure 3.1. Our data flow graphs are similar to Loop Communication Graphs (LCGs) from related work in imperative array fusion (Gao et al., 1993). We name edges after the corresponding variable from the CNF form, and edges which are fusion preventing are drawn with a dash through them (as per the edge labeled `sum1` in figure 3.1). In data flow graphs, we tend to elide the worker functions to combinators when they are not important to the discussion — so we don't show the `(+)` operator on each use of `fold`.

Clusters of operators that are fused into single imperative loops are indicated by dotted lines, and we highlight materialized arrays by drawing them in boxes. In figure 3.1, the variables `xs`, `ys1` and `ys2` are always in boxes, as these are the material input and output arrays of the program. However, in the graph on the far right hand side, `gts` has also been materialized because in this version, the producing and consuming operators (`filter` and `fold`) have not been fused. In figure 3.2, note that the bindings have been split into those that produce scalar values (*sbind*), and those that produce array values (*abind*). These groupings are represented as open and closed arrow-heads in figure 3.1.

Most of our array combinators are standard, and suggestive types are given at the bottom of figure 3.2. The $\texttt{map}_n$ combinator takes a worker function, n arrays of the same length, and applies the worker function to all elements at the same index. As such, it is similar to Haskell's `zipWith`, with an added length restriction on the argument arrays. The `generate` combinator takes an array length and a worker function, and creates a new array by applying the worker to each index. The `gather` combinator takes an array of elements, an array of indices, and produces the array of elements that are positioned at each index. In Haskell, this would be `gather arr ixs = map (index arr) ixs`. The `cross` combinator returns the cartesian product of two arrays.

The exact form of the worker functions is left unspecified, as it is not important for the discussion. We assume workers are pure, can at least compute arithmetic functions of their scalar arguments, and index into arrays in the environment. We also assume that each CNF program considered for fusion is embedded in a larger host program which handles file IO and the like. Workers are additionally restricted so they can only directly reference the *scalar* variables bound by the local CNF program, though they may reference array variables bound by the host program. All access to locally bound array variables is via the formal parameters of array combinators, which ensures that all data dependencies we need to consider for fusion are explicit in the data flow graph.

The `external` binding invokes a host library function that can produce and consume arrays, but not be fused with other combinators. All arrays passed to and returned from host

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

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

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

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

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

Figure 3.2: Combinator normal form

functions are fully materialised. External bindings are explicit *fusion barriers*, which force arrays and scalars to be fully computed before continuing.

Finally, note that `filter` is only one representative size changing operator. We can handle more complex functions such as `unfold` in our framework, but we stick with simple filtering to aid the discussion.

$$
\begin{array}{llll}
\textbf{Size Type} & \tau & ::= k & \text{(size variable)} \\
 & & \mid \quad \tau \times \tau & \text{(cross product)} \\
\\
\textbf{Size Constraint} & C & ::= \mathit{true} & \text{(trivially true)} \\
 & & \mid \quad k = \tau & \text{(equality constraint)} \\
 & & \mid \quad C \wedge C & \text{(conjunction)} \\
\\
\textbf{Size Scheme} & \sigma & ::= \forall \overline{k}.\ \exists \overline{k}.\ (\overline{x : \tau}) \rightarrow (\overline{x : \tau}) &
\end{array}
$$

Figure 3.3: Sizes, Constraints and Schemes

3.3 SIZE INFERENCE

Before performing fusion proper, we must infer the relative sizes of each array in the program. We achieve this with a simple constraint based inference algorithm, which we discuss in this section. Size inference has been previously described in the context of array fusion by Chatterjee (Chatterjee et al., 1991). In constrast to our algorithm, (Chatterjee et al., 1991) does not support size changing functions such as filter. If size inference fails, the programs may still be compiled, but fusion is not performed.

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

3.3.1 *Size Types, Constraints and Schemes*

Figure 3.3 shows the grammar for size types, constraints and schemes. A size scheme is like a type constraint from Hindley-Milner type systems, except that it only mentions the size of each input array, instead of the element types as well.

A size may either be a variable k or a cross product of two sizes. We use the latter to represent the result size of the `cross` operator discussed in the previous section. Constraints may either be trivially *true*, an equality $k = \tau$, or a conjunction of two constraints $C \wedge C$. We refer to the trivially true and equality constraints as *atomic constraints*. Size schemes relate the sizes of each input and output array. For example, the size scheme for the `normalize2` example from figure 3.1 is as follows:

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

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

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

```
filterLeft :s ∀k1.∃k2.(xs : k1) → (ys1 : k1, ys2 : k2)
filterLeft xs
  = let ys1 = map (+ 1)   xs
        ys2 = filter even xs
    in (ys1, ys2)
```

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

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

```
filterLeft : ∀ k₁ : Nat. ∃ k₂ : Nat.
  Array k₁ Float → (Array k₁ Float, Array k₂ Float)
```

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

3.3.2 *Constraint Generation*

The rules for constraint generation are shown in figure 3.4. The top level judgment $program :_s \sigma$ assigns a size scheme to a program. It does this by extracting size constraints and then solving them. This rule, along with the constraint solving process is discussed in the next section. The judgment $\Gamma_1 \mid zs \vdash b \rightsquigarrow \Gamma_2 \vdash C$ reads: "Under environment Γ_1, array variable zs binds the result of b, producing a result environment Γ_2 and size constraints C". The remaining judgment that extracts constraints from a list of bindings is similar. The environment Γ has the following grammar:

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

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

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

For example, the final environment and constraints generated for the `normalize2` example from section 3.1 are as follows:

$$\begin{array}{ll} & x : k_{xs},\ gts : k_{gts},\ \exists k_1,\ k_2,\ k_3 \\ \vdash & true \ \wedge\ k_{gts} = k_1 \ \wedge\ true \\ & \quad \wedge\ k_{xs} = k_2 \ \wedge\ k_{ys1} = k_2 \ \wedge\ k_{xs} = k_3 \ \wedge\ k_{ys2} = k_3 \end{array}$$

Rule (SProgram) also characterises the programs we accept: a program is *valid* if and only if $\exists \sigma.\ program\ :_s \sigma$.

3.3.3 *Constraint Solving and Generalization*

The top-level rule in figure 3.4 assigns a size scheme to a program by first extracting size constraints, before solving them and generalizing the result. In the rule, the solving process is indicated by SOLVE, and takes an environment and a constraint set, and produces a solved environment and constraint set. As the constraint solving process is both standard and straightforward, we only describe it informally.

Recall from the previous section that in our generated constraints all the size variables named after program variables are on the left of atomic equality constraints, while all the unification and existential variables are on the right. To solve the constraints, we keep finding pairs of atomic equality constraints where the same variable appears on the left, unify the right of both of these constraints, and apply the resulting substitution to both the environment and original constraints. When there are no more pairs of constraints with the same variable on the left, the constraints are in solved form and we are finished.

During constraint solving, all unification variables occuring in the environment can have other sizes substituted for them. In contrast, the rigid variables marked by the ∃ symbol cannot. For example, consider the constraints for `normalize2` mentioned before:

$$\begin{array}{rl} & x : k_{xs},\ gts : k_{gts},\ \exists k_1,\ k_2,\ k_3 \\ \vdash & true \ \wedge\ k_{gts} = k_1 \ \wedge\ true \\ & \quad \wedge\ k_{xs} = k_2 \ \wedge\ k_{ys1} = k_2 \ \wedge\ k_{xs} = k_3 \ \wedge\ k_{ys2} = k_3 \end{array}$$

Note that k_{xs} is mentioned twice on the right of an atomic equality constraint, so we can substitute k_2 for k_3. Eliminating the duplicates, as well as the trivially *true* terms then yields:

$$\begin{array}{rl} & x : k_{xs},\ gts : k_{gts},\ \exists k_1,\ k_2 \\ \vdash & k_{gts} = k_1 \ \wedge\ k_{xs} = k_2 \ \wedge\ k_{ys1} = k_2 \ \wedge\ k_{ys2} = k_2 \end{array}$$

To produce the final size scheme, we look up the sizes of the input and output variables of the original program from the solved constraints and generalize appropriately. This process is determined by the top-level rule in figure 3.4. In this case, no rigid size variables appear in the result, so we can universally quantify all size variables.

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

$$\boxed{program \ :_s \ \sigma}$$

$$\frac{\begin{array}{l}\{k_i,\ xs_i : k_i\}^{i\leftarrow 1..n} \ \vdash \ \texttt{let}\ bs\ \texttt{in}\ \{ys_j\}^{j\leftarrow 1..m} \ \rightsquigarrow \ \Gamma[ys_j : k'_j]^{j\leftarrow 1..m} \ \vdash \ C \\ (\Gamma',\ C') = \text{SOLVE}(\Gamma,\ C) \quad \{k_i = s_i\}^{i\leftarrow 1..n} \in C' \quad \{k'_j = t_j\}^{j\leftarrow 1..m} \in C' \\ \overline{k_a} = \{k \mid k \in \Gamma'\} \ \cap \ (\bigcup_{i\leftarrow 1..n} \text{fv}(s_i)) \quad \overline{k_e} = \{k \mid \exists k \in \Gamma'\} \ \cap \ (\bigcup_{j\leftarrow 1..m} \text{fv}(t_j)) \quad \{\exists k \notin \Gamma \mid \bigcup_{i\leftarrow 1..n} \text{fv}(s_i)\}\end{array}}{f\ \{xs\}^{i\leftarrow 1..n} = \texttt{let}\ bs\ \texttt{in}\ \{ys\}^{j\leftarrow 1..m} \ :_s \forall\overline{k_a}.\ \exists\overline{k_e}.\ (\{xs_i : s_i\}^{i\leftarrow 1..n}) \to (\{ys_j : t_j\}^{j\leftarrow 1..m})}$$

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

$$\Gamma \ \vdash \ \texttt{let}\ \cdot\ \texttt{in}\ exp \ \rightsquigarrow \ \Gamma \ \vdash \ true \quad (\text{SNil}) \qquad \frac{\Gamma_1 \mid zs \ \vdash \ b \ \rightsquigarrow \ \Gamma_2 \ \vdash \ C_1 \quad \Gamma_2 \ \vdash \ \texttt{let}\ bs\ \texttt{in}\ exp \ \rightsquigarrow \ \Gamma_3 \ \vdash \ C}{\Gamma_1 \ \vdash \ \texttt{let}\ zs \ = b\ ;\ bs\ \texttt{in}\ exp \ \rightsquigarrow \ \Gamma_3 \ \vdash \ C_1 \wedge C_2}$$

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

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

Figure 3.4: Constraint Generation

3.3.4 *Rigid Sizes*

When the environment of our size constraints contains rigid variables (indicated by $\exists k$), we introduce existential quantifiers instead of universal quantifiers into the size scheme. Consider the `filterLeft` program from section 3.3.1

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

The size constraints for this program, already in solved form, are as follows.

$$\begin{array}{ll} & xs : k_{xs},\ ys_1 : k_{ys1},\ \exists k_1,\ ys_2 : k_{ys2},\ k_2 \\ \vdash & k_{ys_1} = k_1 \ \wedge\ k_{ys_2} = k_2 \ \wedge\ k_{xs} = k_2 \end{array}$$

As variable k_2 is marked as rigid, we introduce an existential quantifier for it, producing the size scheme stated earlier:

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

Note that, although Rule (SProgram) from figure 3.4 performs a *generalisation* process, there is no corresponding instantiation rule. The size inference process works on the entire graph at a time, and there is no mechanism for one operator to invoke another. To say this another way, all subgraphs are fully inlined. Recall from section 3.2, that we assume our operator graphs are embedded in a larger host program. We use size information to guide the clustering process, and although the host program can certainly call the operator graph, static size information does not flow across this boundary.

When producing size schemes, we do not permit the arguments of an operator graph to have existentially quantified sizes. This restriction is necessary to reject programs that we cannot statically guarantee will be well sized. For example:

```
bad1 xs  = let flt   = filter p xs
               ys    = map2   f flt xs
           in  ys
```

The above program filters its input array, and then applies `map2` to the filtered version as well as the original array. As the `map2` operators requires both of its arguments to have the same size, `bad1` would only be valid when the predicate `p` is always true. The size constraints are as follows:

$$\begin{aligned} & xs : k_{xs},\ flt : k_{flt},\ \exists k_1,\ ys : k_{ys},\ k_2 \\ \vdash\ \ & k_{flt} = k_1 \ \wedge\ k_{flt} = k_2 \ \wedge\ k_{xs} = k_2 \ \wedge\ k_{ys} = k_2 \end{aligned}$$

Solving this then yields:

$$\begin{aligned} & xs : k_{xs},\ flt : k_{flt},\ \exists k_1,\ ys : k_{ys},\ k_1 \\ \vdash\ \ & k_{flt} = k_1 \ \wedge\ k_{xs} = k_1 \ \wedge\ k_{ys} = k_1 \end{aligned}$$

In this case, Rule (SProgram) does not apply, because the parameter variable xs has size k_1, but k_1 is marked as rigid in the environment (with $\exists k_1$).

As a final example, the following program is ill-sized, because the two filter operators are not guaranteed to produce the same number of elements.

```
 bad2 xs = let as  = filter p1 xs
               bs  = filter p2 xs
               ys  = map2   f  as bs
           in  ys
```

The initial size constraints for this program are:

$$\begin{array}{ll} & xs : k_{xs},\ as : k_{as},\ \exists k_1,\ bs : k_{bs},\ \exists k_2,\ ys : k_{ys},\ k_3 \\ \vdash & k_{as} = k_1 \ \wedge\ k_{bs} = k_2 \ \wedge\ k_{as} = k_3 \ \wedge\ k_{bs} = k_3 \ \wedge\ k_{ys} = k_3 \end{array}$$

To solve these, we note that k_{as} is used twice on the left of an atomic equality constraint, so we substitute k_1 for k_3:

$$\begin{array}{ll} & xs : k_{xs},\ as : k_{as},\ \exists k_1,\ bs : k_{bs},\ \exists k_2,\ ys : k_{ys},\ k_1 \\ \vdash & k_{as} = k_1 \ \wedge\ k_{bs} = k_2 \ \wedge\ k_{bs} = k_1 \ \wedge\ k_{ys} = k_1 \end{array}$$

At this stage we are stuck, because the constraints are not yet in solved form, and we cannot simplify them further. Both k_1 and k_2 are marked as rigid, so we cannot substitute one for the other and produce a single atomic constraint for k_{bs}.

3.3.5 *Iteration Size*

After inferring the size of each array variable, each operator is assigned an *iteration size*, which is the number of iterations needed in the loop which evaluates that operator. For `filter` and other size changing operators, the iteration and result sizes are in general different. For such an operator, we say that the result size is a *descendant* of the iteration size. Conversely, the iteration size is a *parent* of the result size.

This descendant–parent size relation is transitive, so if we filter an array, then filter it again, the size of the result is a descendant of the iteration size of the initial filter. This relation arises naturally from Data Flow Fusion (Lippmeier et al., 2013), as such an operation would be compiled into a single loop — with an iteration size identical to the size of the input array, and containing two nested if-expressions to perform the two layers of filtering.

Iteration sizes are used to decide which operators can be fused with each other. As in prior work, operators with the same iteration size can be fused. However, in our system we also allow operators of different iteration sizes to be fused, provided those sizes are descendants of the same parent size.

We use T to range over iteration sizes, and write $\bot$ for the case where the iteration size is unknown. The $\bot$ size is needed to handle the `external` operator, as we cannot statically infer its true iteration size, and it cannot be fused with any other operator.

$$\begin{array}{llll} \textbf{Iteration Size} & T & ::= \tau & \text{(known size)} \\ & & \mid\ \ \bot & \text{(unknown size)} \end{array}$$

Once the size constraints have been solved, we can use the following function to compute the iteration size of each binding. In the definition, we use the syntax $\Gamma(xs)$ to find the $xs : k$ element in the environment Γ and return the associated size k. Similarly, we use the syntax $C(k)$ to find the corresponding $k = \tau$ constraint in C and return the associated size type τ.

$$\begin{array}{llll} iter_{\Gamma,C} & : & bind \rightarrow T & \\ iter_{\Gamma,C} & \mid & (z\ = \texttt{fold}\ f\ xs) & = & C(\Gamma(xs)) \\ & \mid & (ys = \texttt{map}_n\ f\ \overline{xs}) & = & C(\Gamma(ys)) \\ & \mid & (ys = \texttt{filter}\ f\ xs) & = & C(\Gamma(xs)) \\ & \mid & (ys = \texttt{generate}\ s\ f) & = & C(\Gamma(ys)) \\ & \mid & (ys = \texttt{gather}\ is\ xs) & = & C(\Gamma(is)) \\ & \mid & (ys = \texttt{cross}\ as\ bs) & = & C(\Gamma(as)) \times C(\Gamma(bs)) \\ & \mid & (ys = \texttt{external}\ \overline{xs}) & = & \bot \end{array}$$

3.3.6 *Transducers*

We define the concept of *transducers* as combinators having a different output size to their iteration size. As with any other combinator, a transducer may fuse with other nodes of the same iteration size, but transducers may also fuse with nodes having iteration size the same as the transducer's output size. For our set of combinators, the only transducer is `filter`.

Looking back at the `normalize2` example, the iteration sizes of the combinators of `gts` and `sum1` are both k_{xs}. The iteration size of `sum2` is k_{gts}, and the filter combinator which produces `gts` is a transducer from k_{xs} to k_{gts}. Even though k_{gts} is not equal to k_{xs}, the three nodes `gts`, `sum1` and `sum2` can all be fused together.

We now define a function *trans*, to find the parent transducer of a combinator application. Since each name is bound to at most one combinator, we abuse terminology here slightly and write *combinator n* when refering to the combinator occuring in the binding of the name *n*. The parent transducer $trans(bs, n)$ of a combinator n has the same output size as n's iteration size, but the two have different iteration sizes.

$$
\begin{array}{lllll}
trans & : & binds \to name \to \{name\} & & \\
trans(bs, o) & & & & \\
 & | & o = \texttt{filter}\ f\ n & \in bs & = & trans'(bs, n) \\
 & | & \text{otherwise} & & = & trans'(bs, o)
\end{array}
$$

$$
\begin{array}{llllll}
trans'(bs, o) & & & & & \\
 & | & o = \texttt{fold}\ f\ n & \in bs & = & \emptyset \\
 & | & o = \texttt{map}_n\ f\ ns & \in bs & = & \bigcup_{x \in ns} trans(bs, x) \\
 & | & o = \texttt{filter}\ f\ n & \in bs & = & \{o\} \\
 & | & o = \texttt{generate}\ s\ f & \in bs & = & \emptyset
\end{array}
$$

$$\begin{array}{llcl}
| \quad o = \texttt{gather}\ i\ d & \in bs & = & trans(bs, i) \\
| \quad o = \texttt{cross}\ a\ b & \in bs & = & \emptyset \\
| \quad o = \texttt{external}\ ins & \in bs & = & \emptyset
\end{array}$$

To determine whether two combinators of different iteration sizes may be fused together, we first find parent or ancestor transducers of the same size, if they exist:

$$\begin{array}{l}
parents \ : \ binds \to name \to name \to \{name \times name\} \\
parents(bs, a, b) \\
\qquad | \ iter_{\Gamma,C}(bs(a)) == iter_{\Gamma,C}(bs(b)) \\
\qquad\qquad = \ \{(a, b)\} \\
\qquad | \ \text{otherwise} \\
\qquad\qquad = \ \{parents(bs, a', b) \mid a' \in trans(bs, a)\} \\
\qquad\qquad \cup \ \{parents(bs, a, b') \mid b' \in trans(bs, b)\}
\end{array}$$

Two combinators a and b of different size may be fused together only if they have parents $(c, d) \in parents(a, b)$, and the combinators and their parents are also fused together. That is, in order for a and b to be fused together, c and d must be fused, a and c must be fused, and d and b must be fused. In the previous example, `sum1` and `sum2` have different iteration size, and their parents are $parents(\texttt{sum1}, \texttt{sum2}) = \{(\texttt{sum1}, \texttt{gts})\})$. In order for `sum1` and `sum2` to be fused together, `sum1` and `gts` must be fused, `sum1` and `sum1` must be fused, and `gts` and `sum2` must be fused. Now we can express the restriction on programs we view as valid for our transformation more formally:

Lemma: sole transducers. If a program p is *valid*, then its bindings will have at most one transducer:

$$\forall p, \sigma, n.\ p :_s \sigma \implies |trans(binds(p), n)| \leq 1$$

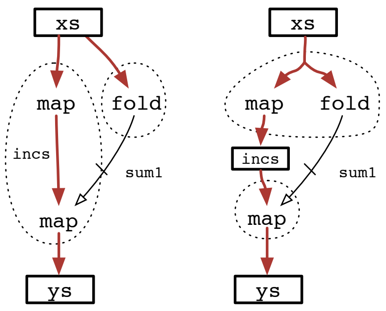


Figure 3.5: Possible clusterings for `normalizeInc`

Lemma: sole parents. For some program p with valid constraints, each pair of names a and b will have at most one pair of parents $parents(a, b)$.

$$\forall p, \sigma, a, b.\, p :_s \sigma \implies |parents(binds(p), a, b)| \leq 1$$

These two lemmas are used in the integer linear programming formulation, when generating the constraints. When fusing two nodes of different iteration size, at most one pair of parents will need to be checked.

3.4 INTEGER LINEAR PROGRAMMING

It is usually possible to cluster a program graph in multiple ways. For example, consider the following simple function:

```
normalizeInc :: Array Int -> Array Int
normalizeInc xs
 = let incs = map  (+1)     us
       sum1 = fold (+) 0    us
       ys   = map  (/ sum1) incs
```

```
  in  ys
```

Two possible clusterings are shown in figure 3.5. One option is to compute `sum1` first and fuse the computation of `incs` and `ys`. Another option is to fuse the computation of `incs` and `sum1` into a single loop, then compute `ys` separately. A third option (not shown) is to compute all results separately, and not perform any fusion.

Which option is better? On current hardware we generally expect the cost of memory access to dominate runtime. The first clustering in figure 3.5 requires two reads from array `xs` and one write to array `ys`. The second requires a single fused read from `xs`, one write to `incs`, a read back from `incs` and a final write to `ys`. From the size constraints of the program we know that all intermediate arrays have the same size, so we expect the first clustering will peform better as it only needs three array accesses instead of four.

For small programs such as `normalizeInc` it is possible to naively enumerate all possible clusterings, select just those that are *valid* with respect to fusion preventing edges, and choose the one that maximises a cost metric such as the number of array accesses needed. However, as the program size increases the number of possible clusterings becomes too large to naively enumerate. For example, Pouchet et al (Pouchet et al., 2010) present a fusion system using the polyhedral model (Pouchet et al., 2011) and report that some simple numeric programs have over 40,000 possible clusterings, with one particular example having 10^{12}.

To deal with the combinatorial explosion in the number of potential clusterings, we instead use an Integer Linear Programming (ILP) formulation. ILP problems are defined as a set of variables, an objective linear function and a set of linear constraints. The integer linear solver finds an assignment to the variables that minimises the objective function, while satisfying all constraints. For the clustering problem we express our constraints regarding fusion preventing edges as linear constraints on the ILP variables, then use the objective function to encode our cost metric. This general approach was first fully described by Megiddo and

Sarkar (Megiddo and Sarkar, 1997), and our main contribution is to extend it to work with size changing operators such as `filter`.

3.4.1 *Dependency Graphs*

A dependency graph represents the data dependencies of the program to be fused, and we use it as an intermediate stage when producing linear constraints for the ILP problem. The dependency graph contains enough information to determine the possible clusterings of the input program, while abstracting away from the exact operators used to compute each intermediate array. The rules for producing dependency graphs are in figure 3.6.

Each binding in the source program becomes a node in the dependency graph. For each intermediate variable, we add a directed edge from the binding that produces a value to all bindings that consume it. Each edge is also marked as either *fusible* or *fusion preventing*. Fusion preventing edges are used when the producer must finish its execution before the consumer node can start. For example, a `fold` operation must complete execution before it can produce the scalar value needed by its consumers. Conversely, the `map` operation produces an output value for each value it consumes, so is marked as fusible.

The `gather` operation is a hybrid: it takes an indices array and an elements array, and for each element in the indices array returns the corresponding data element. This means that gather can be fused with the operation that produces its indices, but not the operation that produces its elements — because those are accessed in a random-access manner.

$$
\begin{array}{lll}
\mathit{nodes} & : & \mathit{program} \to V \\
\mathit{edges} & : & \mathit{program} \to E \\
\mathit{edge} & : & \{\mathit{bind}\} \times \mathit{bind} \to E \\
\mathit{inedge} & : & \{\mathit{bind}\} \times \mathit{name} \times \mathit{name} \to E
\end{array}
$$

$$\mathit{nodes}(bs) \ = \{(\mathit{name}(b), \mathit{iter}_{\Gamma,C}(b)) | b \in bs\}$$

$$\mathit{edges}(bs) \ = \bigcup_{b \in bs} \mathit{edge}(bs, b)$$

$$
\begin{array}{l}
\mathit{edge}(bs, out = \texttt{fold}\ f\ in) \\
\quad = \ \{\mathit{inedge}(bs, out, s) | s \in fv(f)\} \cup \{\mathit{inedge}(bs, out, in)\} \\
\mathit{edge}(bs, out = \texttt{map}\ f\ in) \\
\quad = \ \{\mathit{inedge}(bs, out, s) | s \in fv(f)\} \cup \{\mathit{inedge}(bs, out, in)\} \\
\mathit{edge}(bs, out = \texttt{filter}\ f\ in) \\
\quad = \ \{\mathit{inedge}(bs, out, s) | s \in fv(f)\} \cup \{\mathit{inedge}(bs, out, in)\} \\
\mathit{edge}(bs, out = \texttt{gather}\ data\ indices) \\
\quad = \ \{(out, data, \text{fusion-preventing})\} \cup \{\mathit{inedge}(bs, out, indices)\} \\
\mathit{edge}(bs, out = \texttt{cross}\ a\ b) \\
\quad = \ \{\mathit{inedge}(bs, out, a)\} \cup \{(out, b, \text{fusion-preventing})\} \\
\mathit{edge}(bs, outs = \texttt{external}\ ins) \\
\quad = \ \{(outs, i, \text{fusion-preventing}) | i \in ins\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{inedge}(bs, to, from) \\
\quad | \ \ (from = \texttt{fold}\ f\ s) \in bs \\
\quad = \ (to, from, \text{fusion-preventing}) \\
\quad | \ \ (outs = \texttt{external} \ldots) \in bs \wedge from \in outs \\
\quad = \ (to, outs, \text{fusion-preventing}) \\
\quad | \ \ \mathit{otherwise} \\
\quad = \ (to, from, \text{fusible})
\end{array}
$$

Figure 3.6: Dependency Graphs from Programs

3.4.2 *ILP Variables*

After generating the dependency graph, the next step is to produce a set of linear constraints from this graph. The variables involved in these constraints are split into three groups:

$$x \;:\; \mathit{node} \times \mathit{node} \;\rightarrow\; \mathbb{B}$$

For each pair of nodes with indices i and j we use a boolean variable $x_{i,j}$ which indicates whether those two nodes are fused. We use $x_{i,j} = 0$ when the nodes are fused and $x_{i,j} = 1$ when they are not. Using 0 for the fused case means that the objective function can be a weighted function of the $x_{i,j}$ variables, and minimizing it tends to increase the number of nodes that are fused. The values of these variables are used to construct the final clustering, such that $\forall i, j.\; x_{i,j} = 0 \iff \mathit{cluster}(i) = \mathit{cluster}(j)$.

$$\pi \;:\; \mathit{node} \;\rightarrow\; \mathbb{R}$$

The second group of variables is used to ensure that the clustering is acyclic. This means that for each node in the graph, the dependencies of that node can be executed before the node itself. For each node i, we associate a real π_i such that every node j that depends on i we have $\pi_j > \pi_i$. Our linear constraints will ensure that if two nodes are fused into the same cluster then their π values will be identical — though nodes in different clusters can also have the same π value. Here is an example of a cyclic clustering:

```
cycle xs  = let ys  = map (+1) xs     (C1)
                sum = fold ys         (C2)
                zs  = map (+sum) ys   (C1)
            in  zs
```

There is no fusion-preventing edge directly between the `xs` and `zs` bindings, but there is a fusion-preventing edge between `sum` and `zs`. If the `xs` and `zs` bindings were in the same cluster `C1` and `sum` was in cluster `C2`, there would be a dependency cycle between `C1` and `C2`, and neither could be executed before the other.

$$c \;:\; \textit{node} \quad\to\quad \mathbb{B}$$

The final group of variables is used to help define the cost model encoded by the objective function. Each node is assigned a variable c_i that indicates whether the array the associated binding produces is *fully contracted*. When an array is fully contracted it means that all consumers of that array are fused into the same cluster, so we have $c_i = 0 \iff \forall (i', j) \in E.\ i = i' \implies x_{i,j} = 0$. In the final program, each successive element of a fully contracted array can be stored in a scalar register, rather than requiring an array register or memory storage.

3.4.3 *Linear Constraints*

The constraints we place on the ILP variables are split into four groups: constraints that ensure the clustering is acyclic; constraints that encode fusion preventing edges; constraints on nodes with different iteration sizes, and constraints involving array contraction.

ACYCLIC AND PRECEDENCE-PRESERVING The first group of constraints ensures that the clustering is acyclic:

$$\begin{aligned} x_{i,j} \leq \pi_j - \pi_i \leq N \cdot x_{i,j} &\quad \text{(with an edge from } i \text{ to } j\text{)} \\ -N \cdot x_{i,j} \leq \pi_j - \pi_i \leq N \cdot x_{i,j} &\quad \text{(with no edge from } i \text{ to } j\text{)} \end{aligned}$$

As per Megiddo (Megiddo and Sarkar, 1997) the form of these constraints is determined by whether there is an dependency between nodes i and j. The N value is set to the total number of nodes in the graph.

If there is an edge from node i to j we use the first constraint form shown above. If the two nodes are fused into the same cluster then we have $x_{i,j} = 0$. In this case the constraint simplifies to $0 \leq \pi_j - \pi_i \leq 0$, which forces $\pi_i = \pi_j$. If the two nodes are in *different* clusters then the constraint instead simplifies to $1 \leq \pi_j - \pi_i \leq N$. This means that the difference between the two πs must be at least 1, and less than N. Since there are N nodes, the maximum difference between any two πs would be at most N, so the upper bound of N is large enough to be safely ignored. This means the constraint can roughly be translated to $\pi_i < \pi_j$, which enforces the acyclicity constraint.

If instead there is no edge from node i to j then we use the second constraint form above. As before, if the two nodes are fused into the same cluster then we have $x_{i,j} = 0$, which forces $\pi_i = \pi_j$. If the nodes are in different clusters then the constraint simplifies to $-N \leq \pi_j - \pi_i \leq N$, which effectively puts no constraint on the π values.

FUSION-PREVENTING EDGES As per Megiddo (Megiddo and Sarkar, 1997), if there is a fusion preventing edge between two nodes we add a constraint to ensure the nodes will be placed in different clusters.

$$x_{i,j} = 1$$
(for fusion-preventing edges from i to j)

When combined with the precedence-preserving constraints earlier, setting $x_{i,j} = 1$ also forces $\pi_i < \pi_j$.

FUSION BETWEEN DIFFERENT ITERATION SIZES This group of constraints restricts which nodes can be placed in the same cluster based on their iteration size. The group has three parts. Firstly, either of the two nodes connected by an edge have an unknown ($\perp$) iteration size then they cannot be fused and we set $x_{i,j} = 1$:

$$x_{i,j} = 1$$
(if $\mathit{iter}_{\Gamma,C}(i) = \perp \vee \mathit{iter}_{\Gamma,C}(j) = \perp$)

Secondly, if the two nodes have different iteration sizes and no common parent then they also cannot be fused and we set $x_{i,j} = 1$:

$$\begin{aligned} & x_{i,j} = 1 \\ & (\text{if } iter_{\Gamma,C}(i) \neq iter_{\Gamma,C}(j) \ \wedge \ parents(i,j) = \emptyset) \end{aligned}$$

Finally, if the two nodes had different iteration sizes but *do* have parent transducers of the same size, then the two nodes can be fused if they are fused with their respective parents, and the parents themselves are fused:

$$\begin{aligned} & x_{a,A} \leq x_{a,b} \\ & x_{b,B} \leq x_{a,b} \\ & x_{A,B} \leq x_{a,b} \\ & (\text{if } iter_{\Gamma,C}(a) \neq iter_{\Gamma,C}(b) \ \wedge \ parents(a,b) = \{(A,B)\}) \end{aligned}$$

This last part is the main difference to existing ILP solutions: we allow nodes with different iteration sizes to be fused when their parent transducers are fused. The actual constraints encode a "no more fused than" relationship. For example $x_{a,A} \leq x_{a,b}$ means that nodes a and b can be no more fused than nodes a and A.

As a simple example, consider fusing an operation on filtered data with its generating filter:

```
sum1 = fold (+) 0  xs
gts  = filter (>0) xs
sum2 = fold (+) 0  gts
```

Here $sum1$ and $sum2$ have different iteration sizes and we have that $parents(sum1, sum2) = \{(sum1, gts)\}$. This means that $sum1$ and $sum2$ may only be fused if $sum1$ is fused with $sum1$ (trivial), $sum2$ is fused with gts, and $sum1$ is fused with gts.

ARRAY CONTRACTION The final group gives meaning to the c variables, which represent whether an array is fully contracted:

$$x_{i,j} \quad \leq c_i$$

(for all edges from i)

Recall that an array is fully contracted when all of the consumers are fused with the nodes that produces it, which means that the array does not need to be fully materialized in memory. As per Darte's work on array contraction (Darte and Huard, 2002), we define a variable c_i for each array, and the constraint above ensures that $c_i = 0$ only if $\forall (i', j) \in E.\ i = i' \implies x_{i,j} = 0$. By minimizing c_i in the objective function, we favor solutions that reduce the number of intermediate arrays.

3.4.4 *Objective Function*

The objective function defines the cost model of the program, and the ILP solver will find the clustering that minimizes this function while satisfying the constraints defined in the previous section. The cost model we use in this paper has three components:

- the number of array reads and writes — an abstraction of the amount of memory bandwidth needed by the program;
- the number of intermediate arrays — an abstraction of the amount of intermediate memory needed;
- the number of distinct clusters — an abstraction of the cost of loop management instructions, which maintain loop counters and the like.

The three components of the cost model are a heuristic abstraction of the true cost of executing the program on current hardware. They are ranked in order of importance — so we prefer to minimize the number of array reads and writes over the number of intermediate arrays, and to minimize the number of intermediate arrays over the number of clusters. However, minimizing one component does not necessarily minimize any other. For example, as the fused program executes multiple array operations at the same time, in some cases the clustering that requires the least number of array reads and writes uses more intermediate arrays than strictly necessary.

We encode the ordering of the components of the cost model as different weights in the objective function. First, note that if the program graph contains N combinators (nodes) then there are at most N opportunities for fusion. We then encode the relative cost of loop overhead as weight 1, the cost of an intermediate array as weight N, and the cost of an array read or write as weight N^2. This ensures that no amount of loop overhead reduction can outweigh the benefit of removing an intermediate array, and likewise no number of removed intermediate arrays can outweigh a reduction in the number of array reads or writes. The integer linear program including the objective function is as follows:

$$
\begin{array}{lll}
\text{Minimise} & \Sigma_{(i,j)\in E} W_{i,j} \cdot x_{i,j} & \text{(memory traffic and loop overhead)} \\
& + \ \Sigma_{i \in V} N \cdot c_i & \text{(removing intermediate arrays)} \\
\text{Subject to} & \ldots \ \text{constraints from section 3.4.3} \ \ldots & \\
\text{Where} & W_{i,j} = N^2 \mid \ (i,j) \in E & \\
& \qquad\qquad \text{(fusing } i \text{ and } j \text{ will reduce memory traffic)} & \\
& W_{i,j} = N^2 \mid \ \exists k.(k,i) \in E \wedge (k,j) \in E & \\
& \qquad\qquad (i \text{ and } j \text{ share an input array}) & \\
& W_{i,j} = 1 \ \ \mid \ \mathtt{otherwise} & \\
& \qquad\qquad \text{(the only benefit is loop overhead)} & \\
& N = |V| &
\end{array}
$$

3.4.5 *Fusion-preventing Path Optimisation*

The integer linear program defined in the previous section includes more constraints than strictly necessary to define the valid clusterings. If two nodes have a path between them which includes a fusion preventing edge, then we know up front that they must be placed in different clusters. The following function $possible(a, b)$ determines whether there is any possibility that the two nodes a and b can be fused. Similarly the function $possible'(a, b)$ checks whether there is any possibility that the parents of a and b may be fused.

$$
\begin{aligned}
&possible : && name \times name \longrightarrow \mathbb{B} \\
&possible(a, b) && = \forall p \in path(a, b) \cup path(b, a).\ \text{fusion-preventing} \notin p \\
&possible' : && name \times name \longrightarrow \mathbb{B} \\
&possible'(a, b) && = \exists A, B.\ parents(a, b) = \{A, B\} \land possible(a, b) \\
& && \quad \land\ possible(A, a) \land possible(B, b) \land possible(A, B)
\end{aligned}
$$

With $possible$ and $possible'$ defined, we refine our formulation to only generate constraints between two nodes if there is a chance they may be fused together. Doing this reduces the total number of constraints, and makes the job of the ILP solver easier. The final formulation of the integer linear program follows.

$$
\begin{aligned}
\text{Minimise} \quad & \Sigma_{(i,j)\in E} W_{i,j} \cdot x_{i,j} + \Sigma_{i \in V} N \cdot c_i \\
& (\text{if } possible(i, j)) \\
\text{Subject to} \quad & -N \cdot x_{i,j} \ \leq\ \pi_j - \pi_i \ \leq\ N \cdot x_{i,j} \\
& (\text{if } possible(i, j) \land (i, j) \notin E \land (j, i) \notin E) \\
& x_{i,j} \ \leq\ \pi_j - \pi_i \ \leq\ N \cdot x_{i,j} \\
& (\text{if } possible(i, j) \land (i, j, \text{fusible}) \in E) \\
& \pi_i < \pi_j
\end{aligned}
$$

(if $(i, j, \text{fusion-preventing}) \in E$)

$$x_{i,j} \leq c_i$$

(if $(i, j, \text{fusible}) \in E$)

$$c_i = 1$$

(if $(i, j, \text{fusion-preventing}) \in E$)

$$x_{i,j} = 1$$

(if $\bot \in \{\mathit{iter}_{\Gamma,C}(i), \mathit{iter}_{\Gamma,C}(j)\}$)

$$x_{i',i} \leq x_{i,j}$$

$$x_{j',j} \leq x_{i,j}$$

$$x_{i',j'} \leq x_{i,j}$$

(if $\mathit{iter}_{\Gamma,C}(i) \neq \mathit{iter}_{\Gamma,C}(j) \wedge \mathit{possible}'(i, j)$

$\wedge\ \mathit{parents}(i, j) = \{(i', j')\}$)

$$x_{i,j} = 1$$

(if $\mathit{iter}_{\Gamma,C}(i) \neq \mathit{iter}_{\Gamma,C}(j) \wedge \neg\mathit{possible}'(i, j)$)

Where $W_{ij} = N^2 \mid (i, j) \in E$

(fusing i and j will reduce memory traffic)

$W_{ij} = N^2 \mid \exists k.(k, i) \in E \wedge (k, j) \in E$

(i and j share an input array)

$W_{ij} = 1 \mid$ `otherwise`

(the only benefit is loop overhead)

$N = |V|$

	Unfused		Stream		Megiddo		**Ours**	
	Time	Loops	Time	Loops	Time	Loops	Time	Loops
Normalize2	1.88s	5	1.64s	4	1.82s	3	**1.59s**	**2**
Closest points	3.83s	6	3.33s	5	2.92s	3	**2.92s**	**3**
QuadTree	5.22s	8	5.22s	8	4.72s	2	**4.72s**	**2**

Figure 3.7: Benchmark results

3.5 BENCHMARKS

This section discusses three representative benchmarks, and gives the full ILP program of the first. These benchmarks highlight the main differences between our fusion mechanism and related work. The runtimes of each benchmark are summarized in figure 3.7. We report times for: the unfused case where each operator is assigned to its own cluster; the clustering implied by stream fusion (Coutts et al., 2007); the clustering chosen by Megiddo (Megiddo and Sarkar, 1997), and the clustering chosen by our system.

For each benchmark we report the runtimes of hand-fused C code based on the clustering determined by each algorithm. Although we also have an implementation of our Data Flow Fusion system in terms of a GHC plugin (Lippmeier et al., 2013), we report on hand-fused C code to provide a fair comparison to related work. As mentioned in (Lippmeier et al., 2013), the current Haskell stream fusion mechanism introduces overhead in terms of a large number of duplicate loop counters, which increases register pressure unnecessarily. Hand fusing all code and compiling it with the same compiler (GCC) isolates the true cost of the various clusterings from low level differences in code generation.

We have used both GLPK and CPLEX as external ILP solvers. For small programs such as `normalizeInc`, both solvers produce solutions in under 100ms. For a larger randomly generated example with twenty-five combinators, GLPK took over twenty minutes to produce a solution while the commercial CPLEX solver was able to produce a solution in under one second — which is still quite usable. We will investigate the reason for this wide range in performance in future work.

The benchmark programs are at `https://github.com/amosr/papers/tree/master/2014betterfus` `benches`.

3.5.1 *Normalize2*

To demonstrate the ILP formulation we will use the `normalize2` example from section 3.1, repeated here:

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

We use the final ILP formulation from section 3.4.5. First, we calculate *possible* – that is, the nodes which have no fusion-preventing path between them.

$$\{\{sum1, gts, sum2\}, \{sum1, ys2\}, \{gts, sum2, ys1\}, \{ys1, ys2\}\}$$

The complete ILP program is shown below. Note that in the objective function the weights for $x_{sum1,sum2}$ and $x_{sum2,ys1}$ are both only 1, because they do not share any input arrays.

$$\begin{array}{ll}
\text{Minimise} & 25 \cdot x_{sum1,gts} + 1 \cdot x_{sum1,sum2} + 25 \cdot x_{sum1,ys2} + \\
& 25 \cdot x_{gts,sum2} + 25 \cdot x_{gts,ys1} + 1 \cdot x_{sum2,ys1} + \\
& 25 \cdot x_{ys1,ys2} + 5 \cdot c_{gts} + 5 \cdot c_{ys1} + 5 \cdot c_{ys2} \\
\text{Subject to} & -5 \cdot x_{sum1,gts} \leq \pi_{gts} - \pi_{sum1} \leq 5 \cdot x_{sum1,gts} \\
& -5 \cdot x_{sum1,sum2} \leq \pi_{sum2} - \pi_{sum1} \leq 5 \cdot x_{sum1,sum2} \\
& -5 \cdot x_{sum1,ys2} \leq \pi_{ys2} - \pi_{sum1} \leq 5 \cdot x_{sum1,ys2}
\end{array}$$

$$\begin{array}{lll}
-5 \cdot x_{gts,ys1} & \leq \pi_{ys1} - \pi_{gts} & \leq 5 \cdot x_{gts,ys1} \\
-5 \cdot x_{sum2,ys1} & \leq \pi_{ys1} - \pi_{sum2} & \leq 5 \cdot x_{sum2,ys1} \\
-5 \cdot x_{ys1,ys2} & \leq \pi_{ys2} - \pi_{ys1} & \leq 5 \cdot x_{ys1,ys2} \\
x_{gts,sum2} & \leq \pi_{sum2} - \pi_{gts} & \leq 5 \cdot x_{gts,sum2} \\
 & \pi_{sum1} < \pi_{ys1} & \\
 & \pi_{sum2} < \pi_{ys2} & \\
x_{gts,sum2} & \leq c_{gts} & \\
x_{gts,sum2} & \leq x_{sum1,sum2} & \\
x_{sum1,sum1} & \leq x_{sum1,sum2} & \\
x_{sum1,gts} & \leq x_{sum1,sum2} &
\end{array}$$

One minimal solution to this is:

$$\begin{array}{ll}
x_{sum1,gts},\ x_{sum1,sum1},\ x_{sum1,sum2},\ x_{gts,sum2},\ x_{ys1,ys2} & = 0 \\
x_{sum1,ys2},\ x_{gts,ys1},\ x_{sum2,ys1} & = 1 \\
\pi_{sum1},\ \pi_{gts},\ \pi_{sum2} & = 0 \\
\pi_{ys1},\ \pi_{ys2} & = 1 \\
c_{gts},\ c_{ys1},\ c_{ys2} & = 0
\end{array}$$

This minimal solution is not unique, though in this case the only other minimal solutions use different π values, and denote the same clustering. Looking at just the non-zero variables in the objective function, the value is $25 \cdot x_{sum1,ys2} + 25 \cdot x_{gts,ys1} + 1 \cdot x_{sum2,ys1} = 51$. For illustrative purposes, note that objective function could be reduced by setting $x_{sum1,ys2} = 0$ (fusing $sum1$ and $ys1$), but this conflicts with other constraints. Since $x_{sum1,sum2} = 0$, we require $\pi_{sum1} = \pi_{sum2}$, but also $\pi_{sum2} < \pi_{ys2}$. These constraints cannot be satisfied, so a clustering that fused $sum1$ and $ys2$ would not also permit $sum1$ and $sum2$ to be fused.

We will now compare the clustering produced by our system, with the one implied by stream fusion. As described in (Lippmeier et al., 2013), stream fusion cannot fuse a produced array into multiple consumers, or fuse operators that are not in a producer-consumer relationship. The corresponding values of the x_{ij} variables are:

$$\begin{aligned} x_{gts,sum2} &= 0 \\ x_{sum1,gts}, x_{sum1,sum2}, x_{ys1,ys2}, x_{sum1,ys2}, x_{gts,ys1}, x_{sum2,ys1} &= 1 \end{aligned}$$

We can force this clustering to be applied in our integer linear program by adding the above equations as new constraints. Solving the resulting program then yields:

$$\begin{aligned} \pi_{sum1}, \pi_{gts}, \pi_{sum2} &= 0 \\ \pi_{ys1}, \pi_{ys2} &= 1 \\ c_{gts}, c_{ys1}, c_{ys2} &= 0 \end{aligned}$$

Note that although nodes $sum1$ and $sum2$ have equal π values, they are not fused. Conversely, if two nodes have different π values then they are never fused.

3.5.2 *Closest Points*

The closest points benchmark is a divide-and-conquer algorithm that finds the closest pair of 2-dimensional points in an array. We first find the midpoint along the Y-axis, and filter the remaining points to those above and below the midpoint. We then recursively find the closest pair of points in the two halves, and merge the results. As the filtered points are passed directly to the recursive call, there is no further opportunity to fuse them, and our clustering is the same as returned by Megiddo's algorithm. However, our clustering generates both filtered arrays in a single loop, unlike stream fusion that requires a separate loop for each.

3.5.3 *QuadTree*

The QuadTree benchmark recursively builds a 2-dimensional space partitioning tree from an array of points. At each step the array of points is filtered into four 2-dimensional boxes. As with the closest points algorith, there are no further opportunities for fusing the filtered results, and our clustering is the same as Megiddo's. However, our clustering produces all four filtered results in a single loop, whereas stream fusion requires four loops.

3.5.4 *QuickHull*

The core of the QuickHull algorithm is shown below: given a line and an array of points, we filter the points to those above the line, and also find the point farthest from that line.

```
hull :: (Point,Point) -> Array Point -> Array Point
hull line@(l,r) pts
 = let pts' = filter (above   line) pts
       ma   = fold   (maxFrom line) pts'
   in (hull (l, ma) pts') ++ (hull (ma, r) pts')
```

Stream fusion cannot fuse the `pts'` and `ma` bindings because `pts'` is referred to multiple times and thus cannot be inlined. Megiddo's algorithm also cannot fuse the two bindings because their iteration sizes are different. If the `ma` binding was rewritten to operate over the `pts` array instead of `pts'`, Megiddo's formulation would be able to fuse the two, and the overall program would give the same result. However, this performance behavior is counter intuitive because `pts'` is likely to be smaller than `pts`, so in an unfused program the original version would be faster. Our system fuses both versions.

3.6 RELATED WORK

The idea of using integer linear programming to cluster an operator graph for array fusion was first fully described by Megiddo and Sarkar (Megiddo and Sarkar, 1997) (1999). A simpler formulation, supporting only loops of the same iteration size, but optimizing for array contraction, was then described by Darte and Huard (Darte and Huard, 2002) (2002). Both algorithms were developed in the context of imperative languages (Fortran) and are based around a Loop Dependence Graph (LDG). In a LDG the nodes represent imperative loops, and the edges indicate which loops may or may not be fused. Although this work was developed in a context of imperative programming, the conceptual framework and algorithms are language agnostic. In earlier work, Chatterjee (Chatterjee, 1993) (1991) mentioned that ILP can be used to schedule a data flow graph, though did not give a complete formulation. Our system extends the prior ILP approaches with support for size changing operators such as `filter`.

In the loop fusion literature, the ILP approach is considered "optimal" because it can find the clustering that minimizes a global cost metric. In our case the metric is defined by the objective function of section 3.4.4. Besides optimal algorithms, there are also heuristic approaches. For example, Gao, Olsen and Sarkar (Gao et al., 1993) use the maxflow-mincut algorithm to try to maximize the number of fused edges in the LDG. Kennedy (Kennedy, 2001) describes another greedy approach which tries to maximize the reuse of intermediate arrays, and Song (Song et al., 2004) tries to reduce memory references.

Greedy and heuristic approaches that operate on lists of bindings rather than the graph, such as Rompf (Rompf et al., 2013), can find optimal clusterings in some cases, but are subject to changes in the order of bindings. In these cases, reordering bindings can produce a different clustering, leading to unpredictable runtime performance.

Darte (Darte, 1999) formalizes the algorithmic complexity of various loop fusion problems and shows that globally minimizing most useful cost metrics is NP-complete. Our ILP formulation itself is NP-hard, though in practice we have not yet found this to be a problem.

Recent literature on array fusion for imperative languages largely focuses on the polyhedral model. This is an algebraic representation imperative loop nests and transformations on them, including fusion transformations. Polyhedral systems (Pouchet et al., 2011) are able to express *all possible* distinct loop transformations where the array indices, conditionals and loop bounds are affine functions of the surrounding loop indices. However, the polyhedral model is not applicable to (or intended for) one dimensional filter-like operations where the size of the result array depends on the source data. Recent work extends the polyhedral model to support arbitrary indexing (Venkat et al., 2014), as well as conditional control flow that is predicated on arbitrary (ie, non-affine) functions of the loop indices (Benabderrahmane et al., 2010). However, the indices used to write into the destination array must still be computed with affine functions.

Ultimately, the job of an array fusion system is to make the program go as fast as possible on the available hardware. Although the cost metrics of "optimal" fusion systems try to model the performance behavior of this hardware, it is not practical to encode the intricacies of all available hardware in a single compiler implementation. Iterative compilation approaches such as (Ashby and O'Boyle, 2006) instead enumerate many possible clusterings, use a cost metric to rank them, and perform benchmark runs to identify which clustering actually performs the best. An ILP formulation like ours naturally supports this model, as the integer constraints define the available clusterings, and the objective function can be used to rank them.

3.7 CONCLUSION

We have shown that using integer linear programming to find clusterings is able to produce predictably better clusterings than heuristic and greedy approaches. Having a predictable clustering algorithm is particularly important when combined with other transformations, such as the vectorisation done by Data Parallel Haskell (Chakravarty et al., 2007). In the case of Data Parallel Haskell, the actual combinators to be clustered are not written directly by the programmer, so being able to find a good clustering without the programmer tweaking the combinators is important.

One obvious shortcoming of this work is the limited selection of combinators. Other combinators can currently be used as external computations, but this is not ideal as they will not be fused. Combinators such as `append`, `scan` and `slice` would be simple to add.

Implementing `length` is particularly interesting, as it does not require the array to be manifest, but does require some array with the same rate to be manifest. For example, finding the length of the output of a filter can only be done after the filter is computed. Once `length` is implemented, more interesting functions such as `reverse` can be implemented as a `generate` followed by a `gather`.

For this work to be useful for Data Parallel Haskell, more combinators are required such as segmented fold and segmented map, which operate on segmented representations of nested arrays. Rate inference will have to be adapted to handle segmented arrays, possibly using an inner and outer rates. For example, segmented fold takes a segmented array of inner and outer rate, and returns a single array of the outer rate. Similarly, segmented append takes two segmented arrays with the same outer rate, but different inner rates, and returns a segmented array with a new inner rate and the same outer rate.

Data flow fusion, which generates the sequential loops, must also be extended with these extra combinators. Data flow fusion can also be extended to generate parallel loops for most combinators by simply splitting the workload among processors. Merging the output of each loop can be performed by concatenating the results of filters, and merging the folds, assuming the fold operation is associative.

Another interesting possibility is combinators where the output order is not important. There are two orderings for cross product: one that requires the second array to be manifest, the other requires the first array to be manifest. It may be worthwhile to add a separate cross combinator that does not ensure which ordering is used, but instead uses the ordering that produces the best clustering. This could be achieved in the integer linear program by requiring that at least one of the cross combinator's inputs are not fused together.

BIBLIOGRAPHY

Ashby, T. J. and O'Boyle, M. F. P. (2006). Iterative collective loop fusion. In *CC: Compiler Construction.*

Benabderrahmane, M.-W., Pouchet, L.-N., Cohen, A., and Bastoul, C. (2010). The polyhedral model is more widely applicable than you think. In *CC: Compiler Construction.*

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.*

Chakravarty, M., Leshchinskiy, R., Jones, S., Keller, G., and Marlow, S. (2007). Data Parallel Haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multi-core programming*. ACM.

Chatterjee, S. (1993). Compiling nested data-parallel programs for shared-memory multiprocessors. *TOPLAS: Transactions on Programming Languages and Systems*, 15(3).

Chatterjee, S., Blelloch, G. E., and Fisher, A. L. (1991). Size and access inference for data-parallel programs. In *PLDI: Programming Language Design and Implementation.*

Chen, D.-K., Su, H.-M., and Yew, P.-C. (1990). *The impact of synchronization and granularity on parallel systems*, volume 18. ACM.

Chiba, Y., Aoto, T., and Toyama, Y. (2010). Program transformation templates for tupling based on term rewriting. *IEICE TRANSACTIONS on Information and Systems*, 93(5):963–973.

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*.

Darte, A. (1999). On the complexity of loop fusion. In *PACT: Parallel Architectures and Compilation Techniques*.

Darte, A. and Huard, G. (2002). New results on array contraction. In *ASAP*.

Gao, G., Olsen, R., Sarkar, V., and Thekkath, R. (1993). Collective loop fusion for array contraction. *Languages and Compilers for Parallel Computing*.

Geilen, M. and Basten, T. (2003). Requirements on the execution of kahn process networks. In *Programming languages and systems*.

Gill, A., Launchbury, J., and Peyton Jones, S. L. (1993). A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93. ACM.

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.

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.*

Keller, G., Chakravarty, M. M., Leshchinskiy, R., Peyton Jones, S., and Lippmeier, B. (2010). Regular, shape-polymorphic, parallel arrays in haskell. *ACM Sigplan Notices.*

Kennedy, K. (2001). Fast greedy weighted fusion. *International Journal of Parallel Programming*, 29(5).

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.*

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

Megiddo, N. and Sarkar, V. (1997). Optimal weighted loop fusion for parallel programs. In *SPAA: Symposium on Parallel Algorithms and Architectures.*

Neumann, T. (2011). Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550.

Odersky, M., Sulzmann, M., and Wehr, M. (1999). Type inference with constrained types. *TAPOS.*

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

Pouchet, L.-N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., and Sadayappan, P. (2010). Combined iterative and model-driven optimization in an automatic parallelization framework. In *SC: High Performance Computing, Networking, Storage and Analysis*.

Pouchet, L.-N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P., and Vasilache, N. (2011). Loop transformations: convexity, pruning and optimization. In *POPL: Principles of Programming Languages*.

Robinson, A., Lippmeier, B., and Keller, G. (2014). Fusing filters with integer linear programming. In *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*, pages 53–62. ACM.

Rompf, T., Sujeeth, A. K., Amin, N., Brown, K. J., Jovanovic, V., Lee, H., Jonnalagedda, M., Olukotun, K., and Odersky, M. (2013). Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *ACM SIGPLAN Notices*. ACM.

Song, Y., Xu, R., Wang, C., and Li, Z. (2004). Improving data locality by array contraction. *Computers, IEEE Transactions on*.

Venkat, A., Shantharam, M., Hall, M. W., and Strout, M. M. (2014). Non-affine extensions to polyhedral code generation. In *CGO: Code Generation and Optimization*.