CHAPTER 1

CLUSTERING FOR ARRAY-BACKED STREAMS

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

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

The contributions of this chapter are:

- We identify an opportunity for improvement over existing imperative clustering algorithms, which do not allow *size-changing operators* such as `filter` to be assigned to the same cluster as operations that consume the output array (Section 1.1);

- We extend the clustering algorithm of Megiddo and Sarkar (1997) and Darte and Huard (2002) with support for size-changing operators. In our system, size-changing operators can be assigned to the same cluster as operations that process their input and output arrays (Section 1.4);

- 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 1.4.5);

- Our constraint system encodes the cost model as a total ordering on the cost of clusterings, expressed using weights on the integer linear program. For example, we encode that memory traffic is more expensive than the overhead of performing a separate pass, so given a choice between the two, memory traffic will be reduced (Section 1.4.4);

- We present benchmarks of our algorithm applied to several common programming patterns. Our algorithm is complete and finds the optimal clustering for the chosen cost model, which yields good results in practice. A cost model which maps exactly to program runtime performance is infeasible in general (Section 1.5).

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

1.1 CLUSTERING WITH FILTERS

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

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

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

Figure 1.1 shows three cluster diagrams for `normalize2`, with each clustering produced by a different clustering algorithm. A cluster diagram is an extended version of the dependency graphs we have already seen; we explain the details in Section 1.2. The leftmost diagram shows how we have to break this program up to execute each part, assuming we use the pull stream model from **??**. With pull streams we cannot compute the sums or the maps concurrently, so we end up with four loops, denoted by dotted lines in the diagram; only the filter operation is combined with the subsequent fold. If we wrote this program to use stream fusion (Coutts et al., 2007), which is a form of pull-based shortcut fusion, we would end up with the same clustering.

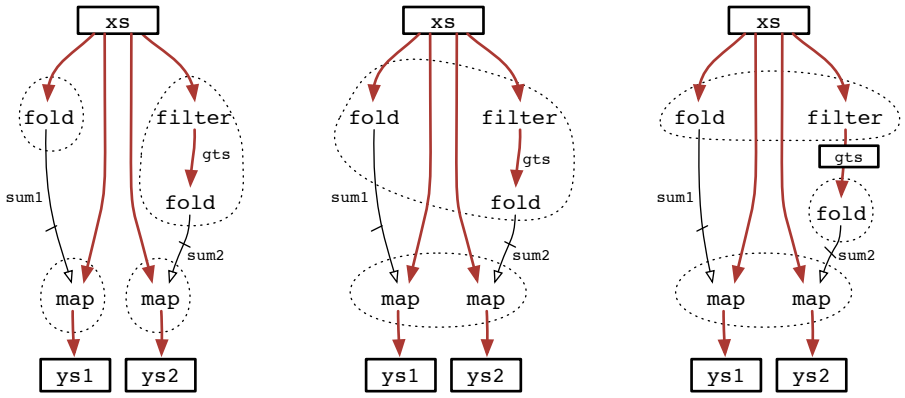

Figure 1.1: Clusterings for `normalize2`: with pull streams; our system; best imperative system

The rightmost diagram in Figure 1.1 shows the clustering determined by the best existing ILP approach for imperative array-based loop fusion. To obtain this clustering, we first implemented each combinator as a separate imperative loop, shown in Listing 1.1. Imperative clustering algorithms, such as Megiddo and Sarkar (1997), only cluster together loops of the same iteration size. In the imperative code, the loop that performs the fold over `gts` has an iteration size of `gts_length`, while all the other loops have an iteration size of `xs_length`. The final value of `gts_length` is not known until the loop that performs the filter completes, so there is a fusion-preventing dependency between the loop that performs the filter and the loop that performs the fold over `gts`, as well as having different iteration sizes. The low-level imperative details obscure the high-level meaning of the program, and complicate fusing the filter operation with the subsequent fold.

Our approach, shown in the middle of Figure 1.1, produces the optimal clustering in this case: one loop for the filter and folds, another for the maps. For this example, we could execute each cluster as either push streams (??) or as a process network fused by process fusion (??). In general, a single cluster produced by our algorithm may contain combinators with multiple inputs as well as multiple outputs, so we execute each cluster as a fused process network.

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

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

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

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

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

Listing 1.1: Unfused imperative implementation of `normalize2`

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

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

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

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

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

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

Figure 1.2: Combinator normal form

1.2 COMBINATOR NORMAL FORM

To perform clustering on an input program, the program is expressed in *combinator normal form* (CNF), which is a textual description of the dependency graph. The grammar for CNF is given in Figure 1.2. Syntactically, a CNF program is a restricted Haskell function definition consisting of one or more let-bound array operations.

The `normalize2` example from Section 1.1 is already in CNF; its corresponding cluster diagrams are shown in Figure 1.1. Our cluster diagrams are similar to Loop Dependence Graphs (LDGs) from related work in imperative array fusion (Gao et al., 1993). We name edges after the

corresponding variable from the CNF form, and edges which are fusion preventing are drawn with a dash through them (as per the edge labeled `sum1` in Figure 1.1). In cluster diagrams, as with dependency graphs, we tend to elide the worker functions of combinators when they are not important to the discussion — so we don't show the `(+)` operator on each use of `fold`.

Clusters of operators, which are to be fused into a single pass by process fusion, are indicated by dotted lines, and we highlight materialized arrays by drawing them in boxes. In Figure 1.1, the variables `xs`, `ys1` and `ys2` are always in boxes, as these are the material input and output arrays of the program. In the rightmost cluster diagram, `gts` has also been materialized because in this version, the producing and consuming operators (`filter` and `fold`) have not been fused together. In the grammar given in Figure 1.2, the bindings have been split into those that produce scalar values (*sbind*), and those that produce array values (*abind*). In the cluster diagrams of Figure 1.1, scalar values are represented by open arrowheads, and array values are represented by closed arrowheads.

Most of our array combinators are standard. Although not part of the grammar, we give the type of each combinator at the bottom of Figure 1.2. 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 located at each index. In Haskell, `gather` would be implemented as (`gather arr ixs = map (index arr) ixs`). The `cross` combinator returns the cartesian product of two arrays.

The exact form of the worker functions is left unspecified. We assume that workers are pure, can at least compute arithmetic functions of their scalar arguments, and index into arrays in the environment. We also assume that each CNF program considered for fusion is embedded in a larger host program which handles file IO and the like. Workers are additionally restricted so

they can only directly reference the *scalar* variables bound by the local CNF program, though they may reference array variables bound by the host program. All access to locally bound array variables is via the formal parameters of array combinators, which ensures that all data dependencies we need to consider for fusion are explicit in the dependency graph.

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

Finally, note that `filter` is only one example of a size-changing operator. We can handle other size-changing operators such as `slice` in our framework, but we stick with simple filtering to aid the discussion. We discuss other combinators in Section 1.7.

1.3 SIZE INFERENCE

The array operations in a cluster are fused together into a loop with a specific number of iterations. Array operations that process different sized arrays cannot usually be assigned to the same cluster, because they require different sized loops. Consumers of arrays produced by size-changing operations can be assigned to the same cluster as operations that process different sized arrays, but only in specific circumstances, which we shall see in Section 1.3.6. Before performing clustering, we need to infer the relative sizes of each array in the program, as the sizes determine the relative loop sizes of each array operation, and whether they can be assigned to the same cluster. We use a simple constraint-based inference algorithm. Size inference has been previously described in the context of array fusion by Chatterjee et al. (1991). In constrast to our algorithm, Chatterjee et al. (1991) does not support size-changing functions such as filter.

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

Figure 1.3: Sizes, constraints and schemes

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

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

1.3.1 *Size types, constraints and schemes*

Figure 1.3 shows the grammar for size types, constraints and schemes. A size scheme is like a type scheme from Hindley-Milner style type systems, except that it only mentions the size of each input array, and ignores the element types.

A size may either be a variable k or a cross product of two sizes. We use the latter to represent the result size of the `cross` operator discussed in the previous section. Constraints may either be trivially *true*, an equality $k = \tau$, or a conjunction of two constraints $C \wedge C$. We refer to the trivially true and equality constraints as *atomic constraints*. Size schemes relate the sizes of each input and output array. The `normalize2` example from Figure 1.1, which returns two output arrays of the same size as the input, has the following size scheme:

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

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

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

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

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

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

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

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

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

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

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

Figure 1.4: Constraint generation for size inference

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

```
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.

1.3.2 *Constraint generation*

The rules for constraint generation are shown in Figure 1.4. The first judgment form is written as ($\Gamma_1 \vdash \mathit{lets} \rightsquigarrow \Gamma_2 \vdash C$) and reads: "under environment Γ_1, the bindings in *lets* produce the result environment Γ_2 and size constraints C". The judgment form ($\Gamma_1 \mid zs \vdash b \rightsquigarrow \Gamma_2 \vdash C$)

performs constraint generation for a single binding and reads: "under environment Γ_1, array variable zs binds the result of b, producing a result environment Γ_2 and size constraints C". The environment (Γ) has the following grammar:

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

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

The constraints are generated in a specific form in Figure 1.4, to facilitate the constraint solving process. For each array variable in the program, we generate a new size variable, like size k_{zs} for array variable zs. These new size variables always appear on the *left* of atomic equality constraints. For each array binding, we may also introduce unification or rigid variables; these appear on the *right* of atomic equality constraints.

The final environment and constraints generated for the `normalize2` example from Section 1.1 are as follows, with the program shown on the right:

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

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

To compute the constraints and environment for this example, the input environment given to constraint generation records that the input array `xs` has the corresponding size type k_{xs}. This input environment is described in Section 1.3.3. For each binding, the rules in Figure 1.4 generate a constraint and add any required array and size bindings to the environment. The `sum1` binding, a `fold`, does not bind any array variables and works for any input size, so the corresponding rule leaves the environment as-is and produces a *true* constraint. For the `gts` binding, a `filter`, the size of the output array is unknown. The `filter` rule records the size of the output array by introducing a new size-type variable k_{gts}, as well as an existential variable k_1; the rule also generates the constraint $(k_{gts} = k_1)$. For the `ys1` binding, a `map`, the size of the output array is the same as the input array. The `map` rule introduces a new size variable k_{ys1} to record the size of the output array, and introduces a new unification variable k_2. The rule introduces constraints requiring the new unification variable k_2 to be equal to both the input size variable k_{xs} and the output size variable k_{ys1}. In the constraints, array size variables occur on the left-hand side and unification variables occur on the right-hand side. Constraint generation for the remaining bindings proceeds similarly.

1.3.3 *Constraint solving and generalization*

Figure 1.5 shows the rule for assigning a size scheme to a program. The top-level judgment form $(program :_s \sigma)$ assigns size scheme σ to *program*.

Rule (SProgram) assigns a size scheme to a program by first extracting size constraints, before solving them and generalizing the result. In the rule, Γ_0 is used as the input environment to constraint generation, and is constructed by generating a fresh size variable (k_i) for each input array (xs_i). The environment and constraints produced by constraint generation are named Γ_1 and C_1; these constraints are then solved using the SOLVE function, which we describe soon. The constraints, after being solved, are stored in C_2, and the environment in

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

$$\frac{\Gamma_0 \vdash \texttt{let}\ bs\ \texttt{in}\ \{ys_j\}^{j \leftarrow 1..m} \rightsquigarrow \Gamma_1 \vdash C_1 \qquad (\Gamma_2, C_2) = \text{SOLVE}(\Gamma_1,\ C_1) \qquad \forall k \in \text{fv}(\overline{s}).\ (\exists k) \notin \Gamma_2}{f\ \{xs\}^{i \leftarrow 1..n} = \texttt{let}\ bs\ \texttt{in}\ \{ys\}^{j \leftarrow 1..m} :_s \forall \overline{k_a}.\ \exists \overline{k_e}.\ (\{xs_i : s_i\}^{i \leftarrow 1..n}) \to (\{ys_j : t_j\}^{j \leftarrow 1..m})} \quad \text{(SProgram)}$$

$$\begin{aligned} \text{where} \quad \Gamma_0 &= \{k_i,\ xs_i : k_i\}^{i \leftarrow 1..n} \\ \\ \overline{s} &= \{s_i \mid i \in 1..n \ \wedge\ (k_i = s_i) \in C_2\} \\ \overline{k'} &= \{k'_j \mid j \in 1..m \ \wedge\ (ys_j : k'_j) \in \Gamma_2\} \\ \overline{t} &= \{t_j \mid j \in 1..m \ \wedge\ (k'_j = t_j) \in C_2\} \\ \\ \overline{k_a} &= \{k \mid k \in \Gamma_2 \ \wedge\ k \in \text{fv}(\overline{s})\} \\ \overline{k_e} &= \{k \mid \exists k \in \Gamma_2 \ \wedge\ k \in \text{fv}(\overline{t})\} \end{aligned}$$

Figure 1.5: Constraint solving for size inference

Γ_2. We use the solved constraints to find the size types of the input arrays ($\overline{s}$), and the size types of the output arrays ($\overline{t}$). We perform generalization by adding universal quantifiers for the unification variables mentioned by the types of input arrays ($\overline{k_a}$), and adding existential quantifiers for the existential variables mentioned by the types of output arrays ($\overline{k_e}$). Finally, we require that the types of input arrays do not mention any existential variables; an example of this restriction is shown in Section 1.3.4.

In the rule, the solving process is indicated by SOLVE, and takes an environment and a constraint set, and produces a solved environment and constraint set. As the constraint solving process is both standard and straightforward, we only describe it informally.

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

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

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

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

$$\begin{aligned} & \texttt{xs} : k_{xs},\ \texttt{gts} : k_{gts},\ \exists k_1,\ \texttt{ys1} : k_{ys1},\ k_2,\ \texttt{ys2} : k_{ys2},\ k_3 \\ \vdash\ & k_{gts} = k_1 \wedge k_{xs} = k_2 \wedge k_{ys1} = k_2 \wedge k_{ys2} = k_2 \end{aligned}$$

To produce the final size scheme, we look up the sizes of the input and output variables of the original program from the solved constraints and generalize appropriately. This process is determined by the top-level rule in Figure 1.5. In the case of `normalize2`, no rigid size variables appear in the result, so we can universally quantify all size variables to get the following size scheme:

$$\texttt{normalize2} :_s \forall k_2.(xs : k_2) \to (ys_1 : k_2,\ ys_2 : k_2)$$

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

1.3.4 *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 1.3.1:

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

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

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

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

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

Note that, although rule (SProgram) from Figure 1.5 performs a *generalization* process, there is no corresponding instantiation rule. The size inference process works on the entire graph at a time, and there is no mechanism for one operator to invoke another. To say this another way, all subgraphs are fully inlined. Recall from Section 1.2 that we assume our operator graphs are embedded in a larger host program. We use size information to guide the clustering process, and although the host program can certainly call the operator graph, static size information does not flow across this boundary.

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

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

The above program filters its input array, and then applies `map2` to the filtered version as well as the original array. As the `map2` operator requires both of its arguments to have the same size, `bad1` would only be valid when the predicate `p` is always true. We could execute this program as a process network if we replaced the `map2` operator with `zipWith`, and explicitly read from the input array `xs` twice, as in the two-source version of `partitionAppend` from **??**. The result size of the `zipWith` operator is the smaller of the two input operators; the extra size restriction on `map2` simplifies size inference, as we do not need to introduce the concept of the minimum of size types. The size constraints for `bad1` are as follows:

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

Solving these constraints then yields:

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

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

inference, as a caller cannot guarantee that an input array's size matches the existential size type chosen by the function.

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

```
bad2 xs
 = let flt1 = filter p1 xs
       flt2 = filter p2 xs
       ys   = map2   f  flt1 flt2
   in  ys
```

The initial size constraints for this program are:

$$\begin{aligned} & xs : k_{xs},\ \mathit{flt1} : k_{\mathit{flt1}},\ \exists k_1,\ \mathit{flt2} : k_{\mathit{flt2}},\ \exists k_2,\ ys : k_{ys},\ k_3 \\ \vdash\ & k_{\mathit{flt1}} = k_1 \ \wedge\ k_{\mathit{flt2}} = k_2 \ \wedge\ k_{\mathit{flt1}} = k_3 \ \wedge\ k_{\mathit{flt2}} = k_3 \ \wedge\ k_{ys} = k_3 \end{aligned}$$

To solve these, we note that k_{flt1} is used twice on the left of an atomic equality constraint, so we substitute k_1 for k_3:

$$\begin{aligned} & xs : k_{xs},\ \mathit{flt1} : k_{\mathit{flt1}},\ \exists k_1,\ \mathit{flt2} : k_{\mathit{flt2}},\ \exists k_2,\ ys : k_{ys},\ k_1 \\ \vdash\ & k_{\mathit{flt1}} = k_1 \ \wedge\ k_{\mathit{flt2}} = k_2 \ \wedge\ k_{\mathit{flt2}} = k_1 \ \wedge\ k_{ys} = k_1 \end{aligned}$$

At this stage we are stuck, because the constraints are not yet in solved form, and we cannot simplify them further. Both k_1 and k_2 are marked as rigid, so we cannot substitute one for the other and produce a single atomic constraint for k_{flt2}. The SOLVE function fails to return a solution, and rule (SProgram) cannot apply.

1.3.5 *Iteration size*

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

This descendant–parent size relation is transitive, so if we filter an array, then filter the resulting array, the size of the result is a descendant of the iteration size of the initial filter. This relation arises naturally from the fact that we compile individual clusters into a single process using process fusion (**??**). With process fusion, such an operation would be compiled into a process containing a single loop that pulls from a stream backed by the input array — with an iteration size identical to the size of the input array, and containing two case instructions to perform the two layers of filtering.

Iteration sizes are used to decide which operators can be fused with each other. As in prior work, operators with the same iteration size can be fused. However, in prior work, operators with different iteration size cannot be assigned to the same cluster, as imperative loop fusion systems cannot generally fuse loops of different iteration sizes. In our system, we also allow operators of different iteration sizes to be fused, provided those sizes are descendants of the same parent size.

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

Iteration Size	T	$::= \tau$		(known size)
		$\mid \perp$		(unknown size)

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

Figure 1.6: Computing the iteration size of a binding

Once the size constraints have been solved, we can use the *iter* function in Figure 1.6 to compute the iteration size of each binding. In the definition, we use the syntax $\Gamma(xs)$ to find the $(xs : k)$ element in the environment Γ and return the associated size k. Similarly, we use the syntax $C(k)$ to find the corresponding $(k = \tau)$ constraint in C and return the associated size type τ.

1.3.6 *Transducers and compatible common ancestors*

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

Looking back at the `normalize2` example, the iteration sizes of the filter over `xs`, which produces the array `gts`, and the fold over `xs`, which produces the scalar `sum1`, are both k_{xs}. The iteration size of the fold over `gts`, which produces the scalar `sum2`, is k_{gts}, and the filter

$$
\begin{array}{lll}
\mathit{trans} & : & \{\mathit{bind}\} \to \mathit{name} \to \{\mathit{name}\} \\
\mathit{trans}(bs, o) & & \\
\quad | & o = \texttt{filter}\ f\ n \quad \in bs & = \quad \mathit{trans}'(bs, n) \\
\quad | & \text{otherwise} & = \quad \mathit{trans}'(bs, o) \\
 & & \\
\mathit{trans}'(bs, o) & & \\
\quad | & o = \texttt{fold}\ f\ n \quad \in bs & = \quad \emptyset \\
\quad | & o = \texttt{map}_n\ f\ ns \quad \in bs & = \quad \bigcup_{x \in ns} \mathit{trans}(bs, x) \\
\quad | & o = \texttt{filter}\ f\ n \quad \in bs & = \quad \{o\} \\
\quad | & o = \texttt{generate}\ s\ f \quad \in bs & = \quad \emptyset \\
\quad | & o = \texttt{gather}\ i\ d \quad \in bs & = \quad \mathit{trans}(bs, i) \\
\quad | & o = \texttt{cross}\ a\ b \quad \in bs & = \quad \emptyset \\
\quad | & o = \texttt{external}\ \mathit{ins} \quad \in bs & = \quad \emptyset
\end{array}
$$

Figure 1.7: Finding the parent transducers of a combinator

combinator which produces `gts` is a transducer from k_{xs} to k_{gts}. Even though k_{gts} is distinct from k_{xs}, the three combinators which produce `gts`, `sum1` and `sum2` can all be fused together.

Figure 1.7 defines a function *trans*, to find the parent transducer of a combinator application. Since each name is bound to at most one combinator, we abuse terminology here slightly and write *combinator n* when refering to the combinator occuring in the binding of the name n. The parent transducer $\mathit{trans}(bs, n)$ of a combinator n has the same output size as n's iteration size, but the two have different iteration sizes. Although the input program's dependency graph forms a directed acyclic graph, the relationship between the combinators in the graph and each combinator's respective parent transducer, if it has one, forms a forest.

With the *trans* function, we can express the restriction on programs we view as valid for clustering as the following:

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

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

Intuitively, by inspecting the definition of *trans′* and performing case analysis on the combinator binding, we see that only the $\texttt{map}_n$ clause in *trans′* can return multiple transducers, and only the `filter` clause returns a singleton set containing a transducer. Since the constraint generation for $\texttt{map}_n$ requires all inputs to have the same size, the inputs will also have the same transducer. If the inputs had different transducers, then their size would be generated by different filters, and each would have its own separate existential variable as a size type, not fulfilling the same-size requirement for $\texttt{map}_n$.

To determine whether two combinators of different iteration sizes may be fused together, Figure 1.8 defines the *concestors* function, which finds the pair of most recent common ancestor transducers such that both ancestors have the same iteration size. In the field of biological systematics, the term *concestor* is defined as the most recent common ancestor; here, we define the *compatible concestors* of a transducer to be the pair of most recent common ancestors with the same iteration size. In the definition of *concestors*, we use the syntax $(name \times name)_\perp$ to denote an optional pair of names. Two combinators a and b of different size may be fused together only if they have compatible concestors $(c, d) \in concestors(a, b)$, and the combinators and their compatible concestors are also fused together. That is, in order for a and b to be fused together, c and d must be fused, a and c must be fused, and d and b must be fused. If the two combinators have the same iteration size, the two compatible concestors will be the combinators themselves, and the above requirements for fusing different sized combinators are trivially satisfied, since a combinator is always fused with itself.

The definition of *concestors′* returns the pair of compatible concestors as well as the distance, determined by counting how many other ancestor transducers there are between the combinator and the compatible concestors. The *closest* function compares the distances of two pairs of compatible concestors and chooses the closest pair. The *increment* function increases the distance by one.

$$
\begin{aligned}
&\mathit{concestors} \;\; \{\mathit{bind}\} \to \mathit{name} \to \mathit{name} \to (\mathit{name} \times \mathit{name})_\perp \\
&\mathit{concestors}(bs, a, b) \\
&\qquad \mid \; (p_a, p_b, d) \in \mathit{concestors}'(bs, a, b) \\
&\qquad\qquad = \; \{(a, b)\} \\
&\qquad \mid \; \text{otherwise} \\
&\qquad\qquad = \; \perp
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{concestors}' \;\; \{\mathit{bind}\} \to \mathit{name} \to \mathit{name} \to (\mathit{name} \times \mathit{name} \times \mathbb{N})_\perp \\
&\mathit{concestors}'(bs, a, b) \\
&\qquad \mid \; \mathit{iter}_{\Gamma,C}(bs(a)) == \mathit{iter}_{\Gamma,C}(bs(b)) \\
&\qquad\qquad = \; \{(a, b, 0)\} \\
&\qquad \mid \; a' \in \mathit{trans}(bs, a),\; p_a \in \mathit{concestors}'(bs, a', b) \\
&\qquad , \; b' \in \mathit{trans}(bs, b),\; p_b \in \mathit{concestors}'(bs, a, b') \\
&\qquad\qquad = \; \mathit{increment}(\mathit{closest}(p_a, p_b)) \\
&\qquad \mid \; a' \in \mathit{trans}(bs, a),\; p_a \in \mathit{concestors}'(bs, a', b) \\
&\qquad\qquad = \; \mathit{increment}(p_a) \\
&\qquad \mid \; b' \in \mathit{trans}(bs, b),\; p_b \in \mathit{concestors}'(bs, a, b') \\
&\qquad\qquad = \; \mathit{increment}(p_b) \\
&\qquad \mid \; \text{otherwise} \\
&\qquad\qquad = \; \perp
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{closest} \;\; : \;\; (\mathit{name} \times \mathit{name} \times \mathbb{N}) \to (\mathit{name} \times \mathit{name} \times \mathbb{N}) \to (\mathit{name} \times \mathit{name} \times \mathbb{N}) \\
&\mathit{closest}((l_a, l_b, l_d), (r_a, r_b, r_d)) \\
&\qquad \mid \; l_d \le r_d \qquad\quad = \; (l_a, l_b, l_d) \\
&\qquad \mid \; \text{otherwise} \qquad = \; (r_a, r_b, r_d)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{increment} : (\mathit{name} \times \mathit{name} \times \mathbb{N}) \to (\mathit{name} \times \mathit{name} \times \mathbb{N}) \\
&\mathit{increment}((a, b, d)) = (a, b, d + 1)
\end{aligned}
$$

Figure 1.8: Finding the compatible concestors, or most recent common ancestors with the same iteration size

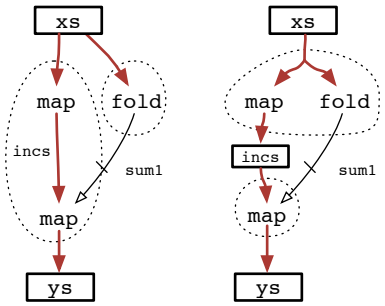

Figure 1.9: Two clusterings for `normalizeInc`

The *trans* function returns only the direct parent transducer, but a single combinator can have multiple ancestor transducers. For a pair of combinators with multiple ancestor transducers, there may be multiple compatible common ancestors, but there can only be one pair of *most recent* compatible common ancestors (compatible concestors), because of the tree structure of ancestor transducers. In general, a pair of differently-sized combinators could be fused together if *any* pair of compatible common ancestors are fused together with the combinators. Because fusing the combinators with any pair of compatible common ancestors requires fusing with the compatible concestors as well, it is sufficient to require the pair of combinators to be fused with just the compatible concestors.

In the previous `normalize2` example, `sum1` consumes the input `xs`, while `sum2` consumes the output of the filter `gts`, which in turn consumes the input `xs`. The two folds have different iteration sizes, and their compatible concestors are *concestors*(`sum1`, `sum2`) = (`sum1`, `gts`). The compatible concestors `sum1` and `gts` both consume the input `xs` and have the same iteration size. In order for `sum1` and `sum2` to be fused together, we require that: `sum1` and `gts` are fused together; `sum1` and `sum1` are fused together, which is trivial as fusion is reflexive; and `gts` and `sum2` are fused together.

1.4 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 1.9. 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 1.9 requires two reads from array `xs` and one write to array `ys`. The second clustering requires a single fused read from `xs`, one write to `incs`, a read back from `incs` and a final write to `ys`. From the size constraints of the program, we know that all intermediate arrays have the same size, so we expect the first clustering will peform better as it only needs three array accesses per element in the input array, instead of four.

For small programs such as `normalizeInc`, it is possible to naively enumerate all possible clusterings, select just those that are *valid* with respect to fusion-preventing edges, and choose the one that maximises a cost metric such as the number of array accesses needed. However, as the program size increases the number of possible clusterings becomes too large to naively enu-

merate. For example, Pouchet et al. (2010, 2011) present a fusion system using the polyhedral model and report that some simple numeric programs have over 40,000 possible clusterings, with one particular example having 10^{12} clusterings.

To deal with the combinatorial explosion in the number of potential clusterings, we instead use an integer linear programming (ILP) formulation. ILP problems are defined as a set of variables, an objective linear function and a set of linear constraints. The integer linear solver finds an assignment to the variables that minimises the objective function, while satisfying all constraints. For the clustering problem, we express our constraints regarding fusion-preventing edges as linear constraints on the ILP variables, then use the objective function to encode our cost metric. This general approach was first fully described by Megiddo and Sarkar (1997); our main contribution is to extend their formulation to work with size-changing operators such as `filter`.

1.4.1 *Dependency graphs*

A dependency graph represents the data dependencies of the program to be fused, and we use it as an intermediate stage when producing linear constraints for the ILP problem. The dependency graph contains enough information to determine the possible clusterings of the input program, while abstracting away from the exact operators used to compute each intermediate array. The rules for producing dependency graphs are shown in Figure 1.10.

Each binding in the source program becomes a node (V) in the dependency graph. For each intermediate variable, we add a directed edge (E) from the binding that produces a value to all bindings that consume it. Each edge is also marked as either *fusible* or *fusion-preventing*. 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

$$
\begin{array}{lll}
V & ::= & (name \times T) \\
E & ::= & (name \times name \times E_T) \\
E_T & ::= & \text{fusible} \mid \text{fusion-preventing}
\end{array}
$$

$$
\begin{array}{lll}
nodes & : & program \to \{V\} \\
nodes(bs) & = & \{(name(b), iter_{\Gamma,C}(b)) \mid b \in bs\}
\end{array}
$$

$$
\begin{array}{lll}
edges & : & program \to \{E\} \\
edges(bs) & = & \bigcup_{b \in bs} edge(bs, b)
\end{array}
$$

$$
\begin{array}{lll}
edge & : & \{bind\} \times bind \to \{E\} \\
edge(bs, out = b) & & \\
\quad \mid \texttt{fold} \quad f\ in & & \leftarrow b \\
\quad \mid \texttt{map} \quad f\ in & & \leftarrow b \\
\quad \mid \texttt{filter}\ f\ in & & \leftarrow b \\
\quad = \{inedge(bs, out, s) \mid s \in \text{fv}(f)\} \cup \{inedge(bs, out, in)\} & & \\
\quad \mid \texttt{gather}\ data\ indices & & \leftarrow b \\
\quad = \{(out, data, \text{fusion-preventing})\} \cup \{inedge(bs, out, indices)\} & & \\
\quad \mid \texttt{cross}\ a\ b & & \leftarrow b \\
\quad = \{inedge(bs, out, a)\} \cup \{(out, b, \text{fusion-preventing})\} & & \\
\quad \mid \texttt{external}\ ins & & \leftarrow b \\
\quad = \{(o, i, \text{fusion-preventing}) \mid o \in out,\ i \in ins\} & &
\end{array}
$$

$$
\begin{array}{lll}
inedge & : & \{bind\} \times name \times name \to E \\
inedge(bs, to, from) & & \\
\quad \mid (from = \texttt{fold}\ f\ s) \in bs & & \\
\quad = (to, from, \text{fusion-preventing}) & & \\
\quad \mid (outs = \texttt{external}\ \ldots) \in bs \wedge from \in outs & & \\
\quad = (to, outs, \text{fusion-preventing}) & & \\
\quad \mid otherwise & & \\
\quad = (to, from, \text{fusible}) & &
\end{array}
$$

Figure 1.10: Dependency Graphs from Programs

$$\begin{array}{lcll} x & : & V \times V & \to \mathbb{B} \\ \pi & : & V & \to \mathbb{R} \\ c & : & V & \to \mathbb{B} \end{array}$$

Figure 1.11: Definition of variables in the integer linear program

output value for each value it consumes; as the values are produced incrementally, its edge is marked as fusible.

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

1.4.2 *Integer linear program variables*

After generating the dependency graph, the next step is to produce a set of linear constraints from this graph. The variables involved in these constraints are split into three groups, shown in Figure 1.11.

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

The second group of variables in Figure 1.11, π, is parameterised by a single node. This group of variables is used to ensure that the clustering is acyclic. An acyclic clustering is

necessary to be able to execute the resulting clustering: we need to ensure that for each node in the graph, the dependencies of that node can be executed before the node itself. For each node i, we associate a real number π_i, such that for every node j that depends on i and is not fused with i, we have $\pi_j > \pi_i$. Our linear constraints will ensure that if two nodes are fused into the same cluster, then their π values will be identical — though nodes in different clusters can also have the same π value. Here is an example of a cyclic clustering, with the cluster (C1) or (C2) specified on the right:

```
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 `ys` and `zs` bindings, but there is a fusion-preventing edge between `sum` and `zs`. If the `ys` 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. We constrain the π variables to reject this clustering, by requiring that $\pi_{\texttt{ys}} \leq \pi_{\texttt{sum}} < \pi_{\texttt{zs}}$. Since $\pi_{\texttt{ys}} < \pi_{\texttt{zs}}$, the two cannot be in the same cluster.

The final group of variables in Figure 1.11, c, is parameterised by a single node. This group of variables is used to help define the cost model encoded by the objective function. Each node is assigned a variable c_i that indicates whether the array produced by the associated binding is *fully contracted*. When an array is fully contracted, it means that all consumers of that array are fused into the same cluster, so we have $c_i = 0 \iff (\forall (i', j) \in E.\ i = i' \implies x_{i,j} = 0)$. In the final program, each successive element of a fully contracted array can be stored in a scalar register, rather than requiring an array register or memory storage.

The names of the first two variable groups are standard; we propose the rather strained mnemonics $x_{i,j}$ denotes an e**x**tra loop between i and j; π_i denotes i's **p**osition in the topological

ordering; and c_i denotes that i's output array is fully **c**ontracted. These three variable groups are a mixture of the variables from previous work. Megiddo and Sarkar (1997)'s formulation uses the x and π groups, but does not include the c group; their cost model does not take into account fully contracted arrays. Darte and Huard (2002)'s formulation uses both c and π groups, where they are called k and ρ respectively, but does not include the x group, which is necessary for our formulation of size-changing operations.

1.4.3 *Linear constraints*

We place linear constraints on the integer linear program variables, and split the constraints into four groups: constraints that ensure the clustering is acyclic; constraints that encode fusion-preventing edges; constraints describing when nodes with different iteration sizes can be fused together; and constraints involving array contraction.

Acyclic and precedence-preserving constraints

The first group of constraints ensures that the clustering is acyclic, using the π variable group described earlier:

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

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

If there is an edge from node i to node j, we use the first constraint form shown above. If the two nodes are fused into the same cluster then we have $x_{i,j} = 0$. In this case, the constraint simplifies to $0 \leq \pi_j - \pi_i \leq 0$, which forces $\pi_i = \pi_j$. If the two nodes are in *different* clusters then

the constraint instead simplifies to $1 \leq \pi_j - \pi_i \leq N$. This means that the difference between the two π variables must be at least 1, and less than or equal to N. Since there are N nodes, the maximum number of clusters, when each node is assigned to its own separate cluster, is N clusters. For this clustering the difference between any two π variables would be at most N, so the upper bound of N is large enough to be safely ignored. Ignoring the upper bound, the constraint can roughly be translated to $\pi_i < \pi_j$, which enforces the acyclicity constraint.

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

Fusion-preventing edges

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

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

(for fusion-preventing edges from i to j)

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

Fusion between different iteration sizes

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

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

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

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

$$\begin{aligned} x_{i,j} \quad &= 1 \\ (\text{if } &\mathit{iter}_{\Gamma,C}(i) \neq \mathit{iter}_{\Gamma,C}(j) \ \wedge \ \mathit{concestors}(i,j) = \bot) \end{aligned}$$

Finally, if the two nodes have different iteration sizes but *do* have compatible common ancestors, then the two nodes can be fused together if they are fused with their respective concestors, and the concestors themselves are fused together:

$$\begin{aligned} x_{a,A} \quad &\leq x_{a,b} \\ x_{b,B} \quad &\leq x_{a,b} \\ x_{A,B} \quad &\leq x_{a,b} \\ (\text{if } &\mathit{iter}_{\Gamma,C}(a) \neq \mathit{iter}_{\Gamma,C}(b) \ \wedge \ (A,B) \in \mathit{concestors}(a,b)) \end{aligned}$$

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

As a simple example, consider fusing an operation on filtered data with its generating filter, as in the folds from `normalize2`:

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

Here, `sum1` and `sum2` have different iteration sizes, and their compatible common ancestor transducers are computed to be $\mathit{concestors}(\texttt{sum1}, \texttt{sum2}) = (\texttt{sum1}, \texttt{gts})$. With the above con-

straints, `sum1` and `sum2` may only be fused together if three requirements are satisfied: `sum1` is fused with `sum1` (trivial), `sum2` is fused with `gts`, and `sum1` is fused with `gts`.

Array contraction

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

$$x_{i,j} \quad \leq c_i$$
(for all edges from i)

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

1.4.4 *Objective function*

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

- the number of array reads and writes — an abstraction of the amount of memory bandwidth needed by the program;
- the number of intermediate arrays — an abstraction of the amount of intermediate memory needed;

- the number of distinct clusters — an abstraction of the cost of loop management instructions, which maintain loop counters and the like.

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

We encode the ordering of the components of the cost model as different weights in the objective function. First, note that if the program graph contains N combinators (nodes) then there are at most N opportunities for fusion, and at most N intermediate arrays. We then encode the relative cost of loop overhead as weight 1, the cost of an intermediate array as weight N, and the cost of an array read or write as weight N^2. These coefficients ensure that no amount of loop overhead reduction can outweigh the benefit of removing an intermediate array, and likewise no number of removed intermediate arrays can outweigh a reduction in the number of array reads or writes. The integer linear program including the objective function is shown in Figure 1.12.

1.4.5 *Fusion-preventing path optimisation*

The integer linear program defined in the previous section includes more constraints than strictly necessary to define the valid clusterings. If two nodes have a path between them that includes a fusion-preventing edge, then we know upfront that they must be placed in different

$$
\begin{array}{lll}
\text{Minimise} & \Sigma_{(i,j)\in E} W_{i,j} \cdot x_{i,j} & \text{(memory traffic and loop overhead)} \\
& + \Sigma_{i \in V} N \cdot c_i & \text{(removing intermediate arrays)} \\
\text{Subject to} & \ldots \text{ constraints from Section 1.4.3 } \ldots & \\
\text{Where} & W_{i,j} = N^2 \quad | \quad (i,j) \in E & \\
& & \text{(fusing } i \text{ and } j \text{ will reduce memory traffic)} \\
& W_{i,j} = N^2 \quad | \quad \exists k.(k,i) \in E \wedge (k,j) \in E & \\
& & (i \text{ and } j \text{ share an input array}) \\
& W_{i,j} = 1 \quad | \quad \texttt{otherwise} & \\
& & \text{(the only benefit is loop overhead)} \\
& N = |V| &
\end{array}
$$

Figure 1.12: Integer linear program with objective function

$$
\begin{aligned}
&\mathit{possible} : \mathit{name} \times \mathit{name} \to \mathbb{B} \\
&\mathit{possible}(a,b) = \forall p \in \mathit{path}(a,b) \cup \mathit{path}(b,a).\ \text{fusion-preventing} \notin p \\
&\mathit{possible}' : \mathit{name} \times \mathit{name} \to \mathbb{B} \\
&\mathit{possible}'(a,b) = \exists A, B.\ (A,B) \in \mathit{concestors}(a,b) \wedge \mathit{possible}(a,b) \\
&\qquad\qquad \wedge\ \mathit{possible}(A,a) \wedge \mathit{possible}(B,b) \wedge \mathit{possible}(A,B)
\end{aligned}
$$

Figure 1.13: Definition of *possible* function for checking fusion-preventing paths

clusters. Figure 1.13 defines the function $\mathit{possible}(a,b)$, which determines whether there is any possibility that the two nodes a and b can be fused. Similarly, the function $\mathit{possible}'(a,b)$ checks whether there is any possibility that the compatible common ancestors of a and b may be fused.

With *possible* and $\mathit{possible}'$ defined, we refine our formulation to only generate constraints between two nodes if there is a chance they may be fused together. The final formulation of the integer linear program is shown in Figure 1.14. This refined version generates fewer constraints, and makes the job of the ILP solver easier.

$$
\begin{aligned}
\text{Minimise}\quad & \Sigma_{(i,j)\in E} W_{i,j} \cdot x_{i,j} + \Sigma_{i \in V} N \cdot c_i \\
& (\text{if } \mathit{possible}(i,j)) \\
\text{Subject to}\quad & -N \cdot x_{i,j} \leq \pi_j - \pi_i \leq N \cdot x_{i,j} \\
& (\text{if } \mathit{possible}(i,j) \wedge (i,j) \notin E \wedge (j,i) \notin E) \\
& x_{i,j} \leq \pi_j - \pi_i \leq N \cdot x_{i,j} \\
& (\text{if } \mathit{possible}(i,j) \wedge (i,j,\text{fusible}) \in E) \\
& \pi_i < \pi_j \\
& (\text{if } (i,j,\text{fusion-preventing}) \in E) \\
& x_{i,j} \leq c_i \\
& (\text{if } (i,j,\text{fusible}) \in E) \\
& c_i = 1 \\
& (\text{if } (i,j,\text{fusion-preventing}) \in E) \\
& x_{i,j} = 1 \\
& (\text{if } \bot \in \{\mathit{iter}_{\Gamma,C}(i), \mathit{iter}_{\Gamma,C}(j)\}) \\
& x_{i',i} \leq x_{i,j} \\
& x_{j',j} \leq x_{i,j} \\
& x_{i',j'} \leq x_{i,j} \\
& (\text{if } \mathit{iter}_{\Gamma,C}(i) \neq \mathit{iter}_{\Gamma,C}(j) \wedge \mathit{possible}'(i,j) \\
& \qquad \wedge \mathit{concestors}(i,j) = \{(i',j')\}) \\
& x_{i,j} = 1 \\
& (\text{if } \mathit{iter}_{\Gamma,C}(i) \neq \mathit{iter}_{\Gamma,C}(j) \wedge \neg \mathit{possible}'(i,j)) \\
\text{Where}\quad & W_{ij} = N^2 \mid (i,j) \in E \\
& \qquad\qquad (\text{fusing } i \text{ and } j \text{ will reduce memory traffic}) \\
& W_{ij} = N^2 \mid \exists k.(k,i) \in E \wedge (k,j) \in E \\
& \qquad\qquad (i \text{ and } j \text{ share an input array}) \\
& W_{ij} = 1 \quad \mid \texttt{otherwise} \\
& \qquad\qquad (\text{the only benefit is loop overhead}) \\
& N = |V|
\end{aligned}
$$

Figure 1.14: Integer linear program with fusion-preventing path optimisation

	Unfused		Stream		Megiddo		**Ours**	
	Time	Loops	Time	Loops	Time	Loops	Time	Loops
Normalize2	0.37s	5	0.31s	4	0.34s	3	**0.28s**	**2**
Closest points	7.34s	7	6.86s	6	6.33s	4	**6.33s**	**4**
Quadtree	0.25s	8	0.25s	8	0.11s	2	**0.11s**	**2**
Quickhull	0.43s	4	0.39s	3	0.28s	2	**0.21s**	**1**

Table 1.1: Benchmark results

1.5 BENCHMARKS

This section discusses four representative benchmarks, and gives the full ILP program of the first benchmark. These benchmarks highlight the main differences between our fusion mechanism and related work. The runtimes of each benchmark are summarized in Table 1.1. We report times and the number of clusters for: the unfused case, where each operator is assigned to its own cluster; the clustering implied by pull-based stream fusion (Coutts et al., 2007); the clustering chosen by Megiddo and Sarkar (1997); and the clustering chosen by our system.

For each benchmark, we compute the different clusterings, and compile each cluster using the process fusion implementation described in **??**. Using the same fusion algorithm isolates the true cost of the various clusterings from low-level differences in code generation.

We have used both GLPK 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 hand-fused implementations of the benchmark programs are available at `https://github.com/amosr/papers/tree/master/2014betterfusionforfilters/benches`.

1.5.1 *Normalize2*

To demonstrate the ILP formulation, we will use the `normalize2` example from Section 1.1, repeated here:

```
normalize2 :: Array Int → (Array Int, Array Int)
```

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

We use the ILP formulation with fusion-preventing path optimisation from Section 1.4.5. First, we calculate the *possible* function to find the nodes which have no fusion-preventing path between them. The sets of nodes which can potentially be fused together are as follows:

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

The complete ILP program is shown in Figure 1.15. In the objective function the weights for $x_{sum1,sum2}$ and $x_{sum2,ys1}$ are both only 1, because the respective combinators do not share any input arrays.

One minimal solution to the integer linear program for `normalize2` is given in Figure 1.16. This minimal solution is not unique, though in this case the only other minimal solutions use different π values, and denote the same clustering. Looking at just the non-zero variables in the objective function, the value is $25 \cdot x_{sum1,ys2} + 25 \cdot x_{gts,ys1} + 1 \cdot x_{sum2,ys1} = 51$. For illustrative purposes, note that the objective function could be reduced by setting $x_{sum1,ys2} = 0$ (fusing $sum1$ and $ys1$), but this conflicts with the other constraints. Since $x_{sum1,sum2} = 0$, we require that $\pi_{sum1} = \pi_{sum2}$, as well as $\pi_{sum2} < \pi_{ys2}$. These constraints cannot be satisfied, so a clustering that fused $sum1$ and $ys2$ would not also permit $sum1$ and $sum2$ to be fused.

We will now compare the clustering produced by our system with the one implied by pull-based stream fusion. As we saw in **??**, pull streams do not support distributing an input stream

$$
\begin{aligned}
\text{Minimise}\quad & 25 \cdot x_{sum1,gts} + 1 \cdot x_{sum1,sum2} + 25 \cdot x_{sum1,ys2} + \\
& 25 \cdot x_{gts,sum2} + 25 \cdot x_{gts,ys1} + 1 \cdot x_{sum2,ys1} + \\
& 25 \cdot x_{ys1,ys2} + 5 \cdot c_{gts} + 5 \cdot c_{ys1} + 5 \cdot c_{ys2}
\end{aligned}
$$

$$
\begin{aligned}
\text{Subject to}\quad & -5 \cdot x_{sum1,gts} && \leq \pi_{gts} - \pi_{sum1} && \leq 5 \cdot x_{sum1,gts} \\
& -5 \cdot x_{sum1,sum2} && \leq \pi_{sum2} - \pi_{sum1} && \leq 5 \cdot x_{sum1,sum2} \\
& -5 \cdot x_{sum1,ys2} && \leq \pi_{ys2} - \pi_{sum1} && \leq 5 \cdot x_{sum1,ys2} \\
& -5 \cdot x_{gts,ys1} && \leq \pi_{ys1} - \pi_{gts} && \leq 5 \cdot x_{gts,ys1} \\
& -5 \cdot x_{sum2,ys1} && \leq \pi_{ys1} - \pi_{sum2} && \leq 5 \cdot x_{sum2,ys1} \\
& -5 \cdot x_{ys1,ys2} && \leq \pi_{ys2} - \pi_{ys1} && \leq 5 \cdot x_{ys1,ys2} \\
& x_{gts,sum2} && \leq \pi_{sum2} - \pi_{gts} && \leq 5 \cdot x_{gts,sum2} \\
& && \pi_{sum1} < \pi_{ys1} \\
& && \pi_{sum2} < \pi_{ys2} \\
& x_{gts,sum2} && \leq c_{gts} \\
& x_{gts,sum2} && \leq x_{sum1,sum2} \\
& x_{sum1,sum1} && \leq x_{sum1,sum2} \\
& x_{sum1,gts} && \leq x_{sum1,sum2}
\end{aligned}
$$

Figure 1.15: Complete integer linear program for `normalize2`

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

Figure 1.16: A minimal solution to the integer linear program for `normalize2`

among multiple consumers; likewise, stream fusion does not support fusing an input with multiple consumers into a single loop. The corresponding values of the x_{ij} variables are:

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

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

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

Note that although nodes $sum1$ and $sum2$ have equal π values, they are not fused because their x values are non-zero. Conversely, if two nodes have different π values, they are never fused.

For the stream fusion clustering, the corresponding value of the objective function is:

$$25 \cdot x_{sum1,gts} + 1 \cdot x_{sum1,sum2} + 25 \cdot x_{sum1,ys2} + 25 \cdot x_{gts,ys1} + 1 \cdot x_{sum2,ys1} + 25 \cdot x_{ys1,ys2} = 102.$$

1.5.2 *Closest points*

The closest points benchmark is a divide-and-conquer algorithm that finds the distance between the closest pair of two-dimensional points in an array. We first find the midpoint along the Y-axis, and filter the remaining points to those above and below the midpoint. We then recursively find the closest pair of points in the two halves, and merge the results.

The closest points implementation is shown in Listing 1.2, with our clustering described in the comments. To compute the clustering of this program, we ignore the base case for small arrays and only look at the recursive case. Our formulation does not directly support the `length` operator; we encode the operation to compute the `midy` midpoint as an `external`

```
closestPoints :: Array Point → Double
closestPoints pts
 | length pts < 100
 -- Naive O(n^2) implementation for small arrays
 = closestPointsNaive pts
 | otherwise
 = let -- (external) Midpoint
       midy    = fold (λs (x,y) → s + y) 0 pts / length pts
       -- (cluster 1) Filter above and below
       aboves  = filter (above midy) pts
       belows  = filter (below midy) pts
       -- (external) Recursive 'divide' step
       above'  = closestPoints aboves
       below'  = closestPoints belows
       border  = min above' below'
       -- (cluster 2) Find points near the border to compare against each other
       aboveB  = filter (λp → below (midy - border) p && above border p) pts
       belowB  = filter (λp → above (midy + border) p && below border p) pts
       -- (cluster 3) Merge results for 'conquer' step
       merged  = cross aboveB belowB
       dists   = map distance merged
       mins    = fold min border dists
   in  mins
```

Listing 1.2: Closest points benchmark

combinator when performing clustering. The two recursive calls are also encoded as `external` combinators. As the filtered points in `aboves` and `belows` are passed directly to the recursive call, there is no further opportunity to fuse them, and our clustering is the same as returned by Megiddo's algorithm. However, unlike stream fusion, our clustering fuses the filter combinators for arrays `aboves` and `belows` into a single loop, as well as fusing the filter combinators for arrays `aboveB` and `belowB`.

1.5.3 *Quadtree*

The Quadtree benchmark recursively builds a two-dimensional space partitioning tree from an array of points. We first find the initial bounding-box that all the points fit into, by computing the minimum and maximum of both X and Y axes. Then, at each recursive step, the bounding-box is split into four quadrants, and each point in the array is placed in the array for its corresponding quadrant.

The Quadtree implementation is shown in Listing 1.3. The `initialBounds` definition deconstructs the input points into two arrays containing the X and Y axes using two `map` operations. The `minimum` and `maximum` operations are implemented as folds. Our clustering for `initialBounds` fuses all six operations into a single loop, as does Megiddo's clustering. Stream fusion requires a separate loop for each fold, and would require a separate loop for each `map` operation, as each result is used twice. The Vector library, which implements stream fusion, supports constant-time `unzip` by using a struct-of-arrays representation for unboxed arrays of pairs. To provide a fair comparison, we replace the `map` operations with constant-time `unzip` in the stream fusion clustering for `initialBounds`, leading to four loops in total.

The `go` function performs the recursive loop over the points array. First, it checks whether the input array is empty or contains a single unique point, and returns a leaf node if so. Otherwise, the bounding-box is split into its four quadrants, and the points are partitioned into an

```
quadtree :: Array Point → Quadtree
quadtree pts = go pts initialBounds
 where
  initialBounds
   = let -- (cluster 1.1) Compute initial bounds
         xs = map fst pts
         ys = map snd pts
         x1 = minimum xs
         x2 = maximum xs
         y1 = minimum ys
         y2 = maximum ys
     in (x1, y1, x2, y2)

  go ins box
   -- Non-recursive base cases for empty and small input arrays
   | length ins == 0
   = Nil
   -- Check if bounding-box contains a single point
   | smallbox box
   = LeafArray ins
   | otherwise
   = let -- (external) Split input box into four quadrants
         (b1,b2,b3,b4)    = splitbox box
         -- (cluster 2.1) Partition input points into above quadrants
         p1               = filter (inbox b1) ins
         p2               = filter (inbox b2) ins
         p3               = filter (inbox b3) ins
         p4               = filter (inbox b4) ins
         -- Recurse into each partitioned quadrant separately
     in  Tree (go p1 b1) (go p2 b2) (go p3 b3) (go p4 b4)
```

Listing 1.3: Quadtree benchmark

```
filterMax :: Line → Array Point → (Point, Array Point)
filterMax l pts
 = let -- (cluster 1) Compute filter and maximum
       ptsAnn   = map (λp → (p, distance p l)) pts
       maximAnn = maximumBy (compare `on` snd)  ptsAnn
       aboveAnn = filter ((>0) ∘ snd)           ptsAnn
       above    = map fst                       aboveAnn
   in (fst maximAnn, above)
```

Listing 1.4: Quickhull core (`filterMax`) implementation

array for each quadrant. Our clustering for go fuses all four filters into a single loop, as does Megiddo's clustering. Stream fusion requires a separate loop for each filter, with four loops in total.

1.5.4 *Quickhull*

We previously benchmarked the Quickhull algorithm in the process fusion benchmarks, in **??**. Listing 1.4 shows the implementation of `filterMax`, which forms the core of the Quickhull algorithm. In our previous benchmark, we saw that stream fusion was unable to fuse `filterMax` into a single loop because the `ptsAnn` array is used twice. Stream fusion only fuses the combinators for `aboveAnn` and `above` together, requiring three loops in total.

For `filterMax`, our clustering algorithm produces a single cluster with all combinators fused together. Megiddo's clustering for `filterMax` requires a separate loop for the `map` operation that produces the `above` array, as it is looping over the result of a filter.

1.6 RELATED WORK

The idea of using integer linear programming to cluster an operator graph for array fusion was first fully described by Megiddo and Sarkar (1997). A simpler formulation, supporting

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

In the loop fusion literature, the ILP approach is considered "optimal" because it can find the clustering that minimizes a global cost metric. In our case, the metric is defined by the objective function of Section 1.4.4. Besides optimal algorithms, there are also heuristic approaches. For example, Gao et al. (1993) use the maxflow-mincut algorithm to try to maximize the number of fused edges in the LDG. Kennedy (2001) describes another greedy approach which tries to maximize the reuse of intermediate arrays, and Song et al. (2004) tries to reduce memory references.

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

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

Recent literature on array fusion for imperative languages largely focuses on the polyhedral model. The polyhedral model is an algebraic representation imperative loop nests and transformations on them, including fusion transformations. Polyhedral systems (Pouchet et al.,

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

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

1.7 FUTURE WORK

One obvious shortcoming of our clustering formulation 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. We now present some ideas of how to support common combinators in future work.

The size inference rules for single-input `map` can be re-used for single-input combinators that produce one output element for every input element, such as `postscanl`, `prescanl`, and `indexed`. Extraction combinators, such as `slice`, `take` and `drop`, take a contiguous 'slice' of the

input. These could be implemented with an existential output size similar to the size inference for `filter`. The `tail` combinator, which discards the first element, could be implemented with an existential output size as with (`drop 1`), or perhaps by adding a new size type to denote 'decrementing' a size type.

Implementing `append` would likely involve adding a new size type for appending two size types together, similar to the result size of the `cross` combinator. Although the result size of appending two concrete input arrays is commutative, the loops that generate the two halves of the output cannot generally be interchanged. The size type for appending two inputs therefore should not be commutative. As we saw in **??**, process fusion can fuse appends with one shared input and two different inputs into a single process. To support this clustering, the definition of transducers would have to be modified to allow fusion between nodes with the same size type as one of the inputs and nodes with the same size as the output. It may be simpler to implement `append` by introducing an existential size type for both the iteration size and the output size; however, introducing an existential for the output size hides the fact that appending two size types is an injection.

The `length` combinator is unique, as it does not require the array to be manifest, but does require some array with the same rate to be manifest. For example, finding the length of the output of a filter can only be done after the filter is computed, while finding the length of the output of a map can often be done before the map is computed. Once `length` is implemented, functions such as `reverse` can be implemented as a `generate` followed by a `gather`.

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*.

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*.

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*.

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

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

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

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*.

FIGURES

TABLES

LISTINGS