Fusing Filters with Integer Linear Programming

Amos Robinson[†] Ben Lippmeier[†] Manuel M. T. Chakravarty[†] Gabriele Keller[†]

[†]Computer Science and Engineering
University of New South Wales, Australia
{amosr,benl,chak,keller}@cse.unsw.edu.au

Abstract

In this paper, we present an array fusion technique for functional programs, which is able to handle multi-loop programs containing size altering operations like filter. Recent work on array fusion shows how to extract data-flow graphs from functional programs and compile them into efficient imperative loops. However, the compilation process only handles graphs that can be compiled into *single* loops, and most programs require more. Other work demonstrated how use Integer Linear Programming (ILP) to *cluster* the operators in a general data-flow graph into subgraphs that can be individually fused, but until now this approach did not handle operations like filter, which produce arrays of a different size than their input. The technique we introduce in this paper combines the advantages of both, by extending the existing ILP approach with support for size changing operators, using an external ILP solver to find good clusterings.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; Control structures; Abstract data types

General Terms Languages, Performance

Keywords Arrays, Fusion, Haskell

1. Introduction

To compile collection-oriented array functional array programs, which play a particular important part in parallel programming, into efficient code, it is essential that we can compile the array operations to a small number of imperative loops. Data flow fusion [**?**] addressed this problem by presenting a technique to compile a specific class of data flow programs into single, efficient imperative loops. It improved on existing related array fusion approaches ([**?**], [**?**] as it 1) it fuses programs that use branching data flows where a produced array is consumed by several consumers, and 2) guarantees complete fusion into a single loop for all programs that operate on the same size input data and contain no fusion-preventing dependencies between operators. However, it can only fuse code fragments into one loop that produce a single output array. Approaches based on Integer Linear Programming ([**?**] and Darte [**?**]) do not suffer from this shortcoming. On the other hand, they cannot handle operations like filter, which produce arrays whose size differs from the input arrays. The technique we present in this paper can handle both multi-loop fragments as well as size-altering operations.

To demonstrate the effect of the three approaches, consider the following code snippet:

```
normalize :: Array Int -> Array Int
normalize 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)
```

Two sums are computed, one of all the elements of `xs`, the other of all the ones greater than zero. Since we need to fully compute the sums to proceed with the maps, it is clear that we need at least two separate loops. These are examples of fusion-preventing dependencies, as fold cannot be fused with subsequent operations. Figure **??** shows the data-flow graph of `normalize` and the effect of all three fusion techniques, where the dotted lines represent the loops clusters. Applying data-flow fusion (left-most diagram) result in four distinct loops: one for the fold for the calculation of `sum1`, one for `sum2`, combining the fold and filter, and two for each map. The best existing ILP approach results in the right-most 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. Our approach (diagram in the middle) produces the, for this example, optimal solution: two loops. The first to calculate both sums, the second to calculate the maps.

Our contributions are as follows:

- We extend prior work by Megiddo [**?**] and Darte [**?**], with support for size changing operators. Size changing operators can be clustered with the operators that generate their source arrays, and compiled naturally with data-flow fusion (§**??**).
- We present a simplification to constraint generation that is also applicable to some existing integer linear programming formulations such as Megiddo's, where constraints between two nodes need not be generated if there exists a fusion-preventing path between the two (§**??**).
- Our constraint system also encodes 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, the memory traffic will be reduced (§**??**).

- We present benchmarks of our algorithm applied to several common programming patterns, and to several pathological examples. Our algorithm is complete and yields good results in practice, though if array sizes are unknown, an optimal solution is uncomputable in general. ***TODO:*** *ref*

The reduction of the clustering problem to integer linear programming was previously described by [**?**], though they do not consider length changing operators.

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

2. Combinator Normal Form

Each input program is expressed in *Combinator Normal Form* (CNF), which is a textual description of its data flow graph. The grammar for CNF is in Figure **??**. Both the `normalize` and `normalize2` examples on the previous page are in CNF, and the matching data flow graph for `normalize2` is in Figure **??**. Our data flow graphs are similar to Loop Communication Graphs (LCGs) from related work in imperative array fusion [**?**]. 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 **??**). 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 **??**, 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 **??**, note that the bindings have been split those that produce scalar values (*sbind*), and those that produce array values (*abind*), and these groupings are represented as open and closed arrow-heads in Figure **??**.

Most of our array combinators are standard, and suggestive types are given at the bottom of Figure **??**. 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 were 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 functions are fully materialised. External bindings are explicit *fusion barriers*, which force arrays and scalars to be fully computed before continuing.

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

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

$$
\begin{array}{lll}
\mathit{sbind} & ::= & \texttt{fold}\ \ \mathit{fun}\ \mathit{array}
\end{array}
$$

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

$$
\begin{array}{lll}
\mathit{function} & ::= & \lambda \mathit{scalar}\ldots\ \mathit{array}\ldots \rightarrow \\
 & & \texttt{let}\ \mathit{bind}\ldots \\
 & & \texttt{in}\ (\mathit{scalar}\ldots, \mathit{array}\ldots)
\end{array}
$$

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

Figure 2. Combinator normal form

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.

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 [**?**]. In constrast to our algorithm, [**?**] does not support size changing functions such as filter.

Although our constraint based formulation of size inference is reminiscent of type inference for HM(X) [**?**], there are important differences. Firstly, our type schemes include existential quantifiers, which represent 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 uses 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.

3.1 Size types, constraints and schemes

Figure **??** 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 maybe either be trivially *true*, an equality $\tau = \tau$, or a conjunction of two constraints $C \wedge C$. We refer to the trivially true and equality constraints as

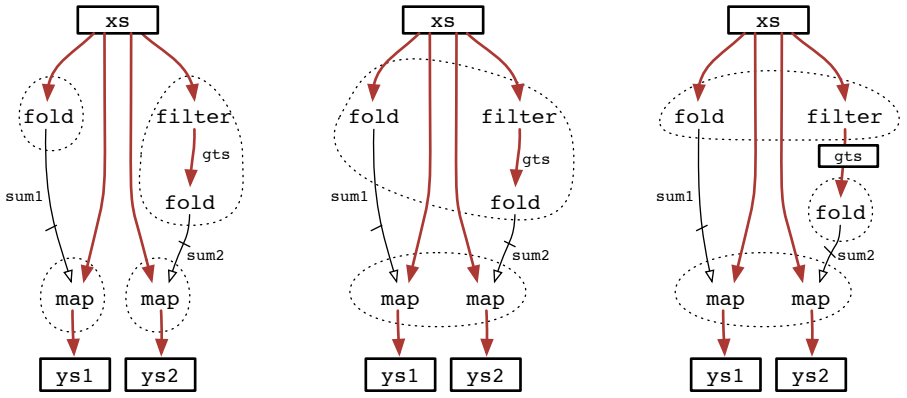

Figure 1. Clusterings for normalize2 example: with stream fusion; our system; best imperative system

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

Figure 3. Sizes, Constraints and Schemes

atomic constraints. Size schemes relate the sizes of each input and output array. For example, the size scheme for the `normalize2` example from Figure **??** is as follows:

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

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

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

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

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

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

3.2 Constraint Generation

The rules for constraint generation are shown in Figure **??**. The top level judgment $function :_s \sigma$ assigns a size scheme to a function. It does this by extracting size constraints and them 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 function.

Note that the constraints are generated in a specific form to make the constraint solving process easy. 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 **??** are as follows:

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

3.3 Constraint Solving and Generalization

The top-level rule in Figure **??** assigns a size scheme to a function by first extracting size constraints, before solving them and generalizing the result. In the rule, the solving process is indicated by SOLVE, and takes and environment and constraint, producing a solved environment and constraint. 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 con-

straints with the same variable on the left then the constraints are in solved form and we are finished.

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

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

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:

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

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

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

3.4 Rigid Sizes

When the environment of our size constraints contains rigid variables (indicate by $\exists k$), we introduce existential quantifiers instead of universal quantifiers into the size scheme. Consider the `filterLeft` function from S **??**

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

The size constraints for this function are as follows, and they are already in solved form.

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

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

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

Note that although Rule (SFun) from Figure **??** 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 §**??** 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
           in  ys    = map2   f flt xs
           in  ys
```

The above function 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:

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

Solving this then yields:

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

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

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

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

The initial size constraints for this function 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 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}.

3.5 Iteration size

After the constraints are validated and equivalence classes generated, each combinator is assigned a *size* as an iteration size – the number of iterations in the loop required to generate the output. It is important to note that for filters, the iteration size is not the output size, but is instead the size of the input. The output size of a filter is, however, a *subsize* of the input's size, as not only is it known to be less than or equal to its input in size, it is also generated depending on the input. The iteration sizes, τ_n, are used to check whether two loops may be fused. Any two iteration sizes in the same equivalence class are the same size, and so are fusible. The difference from previous work is that loops of different iteration sizes *can* be fused, if one is a subsize of the other, and the subsize's *generator* is fused together it as well. Basically, operations on filtered data can be fused with operations on the original data, if it is fused with the filter as well. External computations are treated separately, as they cannot be fused with any other nodes.

T	::=	*size*	(known size)
	|	`external`	(external and unfusible)

Once the constraints are solved, known to be valid, and sorted into equivalence classes, each combinator is assigned a size. Note that for a filter, the size of the output array k_o is some existential that is less than or equal to k_n, but the actual loop size of the *combinator* is equal to k_n. This is because, in order to produce the filtered output, all elements of the input n must be considered.

$$\tau \quad :: \quad binds \to name \to T$$

$$\begin{array}{lllll} \tau_{bs,o} \mid & o = \texttt{fold}\ f\ n & \in bs & = & k_n \\ \mid & o = \texttt{map}_n\ f\ ns & \in bs & = & k_o \end{array}$$

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

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

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

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

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

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

Figure 4. Constraint Generation

$$\begin{array}{llll}
\mid & o = \texttt{filter}\ f\ n \in bs & = & k_n \\
\mid & o = \texttt{generate}\ s\ f \in bs & = & k_o \\
\mid & o = \texttt{gather}\ i\ d \in bs & = & k_i \\
\mid & o = \texttt{cross}\ a\ b \in bs & = & k_a \times k_b \\
\mid & o = \texttt{external}\ ins \in bs & = & \texttt{external}
\end{array}$$

After the generated equality constraints are solved, and sizes are grouped into equivalence classes, the combinators with iteration sizes in the same equivalence class can be fused together.

3.6 Transducers

Unlike previous work, we do allow combinators with different iteration sizes to be fused together. For example, an operation on filtered data may be fused with the filter operation that generates the data, even though the iteration sizes are different. More generally, if the output size of a combinator is different to its iteration size, it is a *transducer* from the iteration size to the output size. As usual, a transducer may fuse with other nodes of the same iteration size, but transducers may also fused with nodes with iteration size the same as the transducer's output size. For our set of combinators, the only transducer is `filter`. ***BL:*** *Do other systems use the concept of transducers?* ***TODO:*** *Explain that a transducer of filter is the filter's parent. so* $trans(trans(n)) \neq n$

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

$$\begin{array}{llll}
trans'(bs, o) & & & \\
\quad \mid & o = \texttt{fold}\ f\ n \in bs & = & \emptyset \\
\quad \mid & o = \texttt{map}_n\ f\ ns \in bs & = & \bigcup_{x\in ns} trans(bs, x) \\
\quad \mid & o = \texttt{filter}\ f\ n \in bs & = & \{o\} \\
\quad \mid & o = \texttt{generate}\ s\ f \in bs & = & \emptyset \\
\quad \mid & o = \texttt{gather}\ i\ d \in bs & = & trans(bs, i) \\
\quad \mid & o = \texttt{cross}\ a\ b \in bs & = & \emptyset \\
\quad \mid & o = \texttt{external}\ ins \in bs & = & \emptyset
\end{array}$$

Lemma: unique transducers. For some bindings bs, each name n will have at most one transducer.

$$valid(\llbracket bs \rrbracket) \implies \forall n.\ |trans(bs, n)| \leq 1$$

Proof: by induction on bs. If $n = \texttt{map}_n\ f\ ns$, then $trans(bs, n) = \bigcup_{x\in ns} trans(bs, x)$. As `map` requires its arguments to have the same size, and `filter` introduces a fresh size for its output, if any of the $trans(bs, x)$ are non-empty, they will refer to the same `filter`. The other cases are trivial.

4. Integer linear program formulation

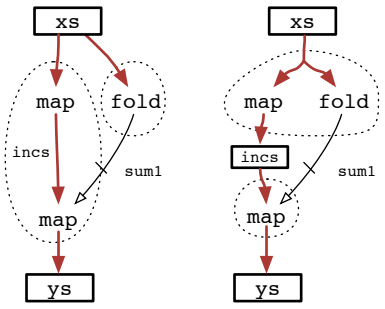

Figure 5. Possible clusterings for `normalizeInc`

TODO: *More justification of why we want to use a heavyweight technology like ILP to do the clustering. Use normalizeInc or one like it to argue that plenty of real programs will admit multiple valid clusterings, so we really want to do this via constraints and an objective function. Compare with prior work on typed fusion that*

only tries to minimise the number of clusters. ***TODO:*** *Also give a quick rundown of what ILP actually is.*

```
normalizeInc :: Array Int -> Array Int
normalizeInc xs
 = let incs = map  (+1)    us      (A1) (B2)
       sum  = fold (+) 0   us      (A1) (B1)
       norm = map  (/ sum) incs    (A2) (B2)
   in  norm
```

For `normalizeInc`, the first approach is to fuse the computation of `incs` and `sum` into a single loop (indicated by `A1`), then use a second loop to compute `norm` (indicated by `A2`). The other approach is to compute `sum` first (`B1`), and fuse the computation of `incs` and `norm` into the second loop (`B2`). For this specific function the second approach seems preferable because it avoids creating the intermediate array `incs`, but for large programs with many operators the optimal way to cluster operators into loops can be entirely non-obvious.

While the different combinators are all implemented differently, certain aspects are shared between all; they operate on input arrays, use scalar variables in their worker functions, and produce output. To simplify creation of the integer linear program formulation, we first convert a function to a dependency graph. The dependency graph has a node for each combinator, and edges between two combinators when one uses another's output. After the graph is created, it is converted to an integer linear program with an objective of the least memory traffic and loops, solved, and the solution converted to a clustering.

When creating the graph, it is important to note that not all loops can be fused together; firstly, loops of different size cannot be fused. Secondly, two loops cannot be fused together if an iteration in one loop relies on the output of a later iteration in another loop. For example, a `map` may not be fused with a `fold` if the `map`'s worker function uses the result of the `fold`. This fusion restriction is encoded as a *fusion-preventing* edge between two combinators. Other edges are *fusible*, and may be fused together. If they are not fused together for whatever reason, the *from* combinator must be scheduled before the *to* combinator.

The dependency graph is translated to an integer linear program. The integer linear program has some integral, boolean and real variables, an objective function to minimise, and a set of constraints that the variables must conform to. Finding a variable assignment that satisfies the constraints and is the minimal objective function is NP-hard, but existing ILP solvers tend to be sufficient for realistic problem sizes. For larger problems, we can find an approximate answer, within say 10% of the optimal answer, which still gives the exact answer for small problems.

4.1 Function → graph

BL: *This is a different sort of graph than before. This is a dependency graph where the nodes are intermediate variables, verses the previous "program graph" where the nodes can also be operators. We need to contrast these properly and add a diagram of a dependency graph.* ***TODO:*** *Put this somewhere: Fusion-preventing dependencies are akin to barrier synchronisations in the Bulk Synchronous model of parallel computation.*

Converting a function in *combinator normal form* to a graph — in fact, a DAG — is quite simple. Each binding becomes a node in the graph. When a binding references other arrays or scalars, there is an edge between those two nodes. Edges may be either *fusible* or *fusion-preventing*. Fusion-preventing edges mean that the entire input node must completely finish its execution before the output node can start executing. For example, `fold`s consume their entire input array before producing a single result, so any references to folds must be fusion-preventing. Conversely, `map`s produce output data for every input element, so may be fused.

The `gather` operation is interesting; it takes an indices array and a data array, and for each element in indices returns that element in data. Thus, `gather` requires random-access in its data array, and is not fusible, but consumes indices linearly and may fuse.

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

$$nodes(bs) \;\; = \{(name(b), \tau_b) | b \in bs\}$$

Each node may simply be the name of its output binding (or bindings, in the case of `external`); as we require names to only be bound once, this is assured to be unique. Creating edges between these nodes is simply when one binding references an earlier one. The only complication is designating edges as *fusible* or *fusion-preventing*.

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

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

$$\begin{array}{l} inedge(bs, to, from) \\ \qquad \mid\ (from = \texttt{fold}\ f\ s) \in bs \\ \qquad = (to, from, fusion\ preventing) \\ \qquad \mid\ (\ldots, from, \ldots = \texttt{external} \ldots) \in bs \\ \qquad = (to, from, fusion\ preventing) \\ \qquad \mid\ otherwise \\ \qquad = (to, from, fusible) \end{array}$$

4.2 Graph → ILP

With the dependency graph now constructed, it must be converted to an integer linear program. The integer linear program will contain some variables, an objective function, and constraints that the variables must conform to. An external solver will find and return a variable assignment that minimises the objective function. With the variable assignment, the graph's nodes are partitioned into a clustering, and then flow fusion[**?**] is used to extract imperative loops for each cluster.

The variables in the ILP formulation can be split into three groups:

$$x \;\; : \;\; node \times node \;\; \to \;\; \mathbb{B}$$

The first and most important group is created for every pair of nodes, and has a boolean value indicating whether the two nodes can be fused. If $x_{ij} = 0$, then i and j are fused together. While it

may seem counterintuitive for 0 to mean fused and 1 unfused, it means the objective function can simply minimise a weighted x_{ij}, and also makes later precedence constraints simpler. The values of these variables are used to construct the partitioning at the end, such that $\forall i, j.\ x_{ij} = 0 \iff cluster(i) = cluster(j)$. ***BL:*** *When we're talking about the dependency graph, when we say "nodes can be fused", it means that the operators that produce these values can be fused – or that the values are produced by the same cluster of operators.*

$$\pi : \quad node \quad \to \quad \mathbb{R}$$

The second group of variables is used to ensure that the clustering is acyclic. For each node i, we associate a real π_i such that every node j after i we have $\pi_j > \pi_i$. This has the effect of requiring an acyclic clustering, as if one node is after another its π must be strictly greater than its predecessor, and the predecessor must be strictly less than the successor. If two nodes are to be fused together, their π values must be the same. ***BL:*** *I don't understand the "acyclic" part. If we don't add these constraints, then how might the clustering turn out to be "cyclic"?*

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

The third, and final, group of variables are purely for minimisation, indicating whether a node's array is *fully contracted*. They are not required for correctness of the clustering, but are used in the objective function. A node's array will be fully contracted — that is, it will be replaced by a scalar — if all nodes that use this array are fused together: $c_i = 0 \iff \forall (i, j) \in E.x_{ij} = 0$. ***BL:*** *Using $\forall$ like this implies that the quantified i isn't the same as the one attached to c_i. Write this with a list comprehension instead.*

4.2.1 Unoptimised version

Before showing the optimised version with certain constraints removed (§**??**), this simpler, unoptimised version is shown. The only difference is that fewer constraints and variables are required in the optimised version, but both versions are equivalent. ***BL:*** *What does "equivalent" mean when the constraints are different?* ***BL:*** *Say what "optimised" means in this case.* ***BL:*** *Give some hints about why the unoptimised version has more constraints.*

Acyclic and precedence-preserving

Minimise …
Subject to …
$x_{ij} \quad \leq \pi_j - \pi_i \leq N x_{ij}$
(an edge from i to j)
$-N x_{ij} \leq \pi_j - \pi_i \leq N x_{ij}$
(no edge from i to j)
…

As per Megiddo[**?**], we look at every pair of nodes i and j.

Whether there is an edge between i and j or not, if the two are fused together, then $x_{ij} = 0$. If $x_{ij} = 0$, then both cases may be simplified to $0 \leq \pi_j - \pi_i \leq 0$, which is equivalent to $\pi_i = \pi_j$. ***BL:*** *Highlight that "fused together" means "in the same cluster", and that operators can be fused even if there are no direct edges between them.*

Otherwise, if the two nodes are not fused together, then $x_{ij} = 1$. If there is an edge between i and j, the constraint simplifies to $1 \leq \pi_j - \pi_i \leq N$, where N is the number of nodes in the graph. This constraint 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 acyclic constraint.

If there is no edge between i and j and the two are not fused together, then $x_{ij} = 1$ and the constraint simplifies to $-N \leq \pi_j - \pi_i \leq N$, which effectively puts no constraint on the π values. Note, however, that other edges may still constrain i or j.

Fusion-preventing edges

Minimise …
Subject to …
$x_{ij} \quad = 1$
(fusion-preventing edges from i to j)
…

As per Megiddo[**?**], for every fusion-preventing edge, we add the constraint $x_{ij} = 1$, so that no fusion can occur. This, together with the precedence constraints above, has the effect of enforcing $\pi_i < \pi_j$.

Fusion between different types ***BL:*** *We haven't said what a "type" is in the context of clustering. Do we mean "rate"? If so, can we use k for the variables instead of τ?*

Minimise …
Subject to …
$x_{ij} \quad = 1$
(if $\tau_i \neq \tau_j \wedge trans_i = \emptyset \wedge trans_j = \emptyset$)
$x_{i'j} \quad \leq x_{ij}$
(if $\tau_i \neq \tau_j \wedge trans_i = \{i'\}$)
$x_{ij'} \quad \leq x_{ij}$
(if $\tau_i \neq \tau_j \wedge trans_j = \{j'\}$)
…

The main difference to existing integer linear programming solutions is that, while nodes with different loop sizes cannot generally be fused together, we do support some fusion of filtered sizes. If two nodes have different sizes or rates, but are both sub-rates of another rate, they *can* be fused together if they are both fused with their generators. ***BL:*** *What is a subrate? How do we determine the subrates of each rate?* ***BL:*** *Highlight why we want to allow this sort of fusion, and why it is specific to data flow fusion.* ***BL:*** *Explain how we are using the idea of transducers.*

Array contraction

Minimise …
Subject to …
$x_{ij} \quad \leq c_i$
(for all edges from i)
…

As per Darte's work on optimising for array contraction[**?**], we define a variable c_i for each array. An array is fully contracted if all its output edges are fused with it. Thus, $c_i = 0$ only if $\forall j | (i, j) \in E.x_{ij} = 0$. By minimising c_i in the objective function, we favour solutions that reduce the number of required intermediate arrays. ***BL:*** *Say what array contraction is, and why we want this constraint.*

4.2.2 The objective function

To find the objective function, note that fusing loops can have three main benefits:

- reducing memory traffic, such as multiple loops reading from the same array;
- removing intermediate arrays, thus reducing the amount of memory required;
- and reducing loop overhead, such as when two loops operate on different arrays of the same size.

However, these benefits cannot be considered in isolation; for example, fusing two loops to reduce loop overhead may remove potential fusion opportunities that reduce memory traffic. When operating on large arrays that do not fit in cache, memory traffic dominates execution time. An excessive number of intermediate arrays can also cause issues if all are live in memory at once, potentially leading to *thrashing*. The benefits of removing loop overhead are least of all; it should be performed if possible, but must never remove opportunities for other fusion. ***TODO:*** *Refer to example code that demonstrates these different opportunities.*

This total ordering can be encoded into an ILP objective function as weights. If the program graph contains N combinators, then there are at most N opportunities for fusion. The encoding of loop overhead is weight 1, removing intermediate arrays is weight N, and reducing memory traffic is 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 memory traffic. Although, it is worth noting that reducing memory traffic does *tend* to remove intermediate arrays, and vice versa.

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

4.3 A note on transitivity

It may seem that we could generate far fewer constraints and rely on transitivity of clustering equality x_{ij}. This means that for each pair i, j we wouldn't generate an x_{ij} and thus not generate those constraints, unless i or j is a `filter`, or arc between i and j. This would help, but also means we can't *minimise* on these removed x_{ij}, so the solution won't count the (rather smaller) benefits of fusing two non-connected nodes.

4.3.1 Fusion-preventing path optimisation

Alternatively: we can say, only generate constraints if there is no *fusion preventing path* between i and j. Here, we generate fewer constraints than originally, but still more than above. We also must preprocess the graph to find such blocking paths. But we still get the minimisation to count non-connected nodes.

$$
\begin{array}{l}
split :: V \times E \to \{\{name\}\} \\
split(vs, es) \\
\quad = \{clusterable(v, vs, es) | v \in vs\} \\
clusterable(v, vs, es) \\
\quad = \{v' | v' \in vs \\
\quad \wedge\ \forall p.p = path(v, v') \vee p = path(v', v) \\
\quad \wedge\ fusion\ preventing \notin p\}
\end{array}
$$

We can simplify the version above and remove many constraints and variables by noting that if there is a fusion-preventing edge between i and j, $x_{ij} = 1$, so the precedence constraint can be simplified to just $\pi_i < \pi_j$, and the x_{ij} variable removed. Likewise with clusters of different, incompatible types. The contraction variable c_i can be removed, if i has any fusion-preventing output edges: any such edges use the array and cannot be fused, thus there is no possibility of the array being contracted away.

$$
\begin{array}{ll}
\text{Minimise} & \ldots \\
\text{Subject to} & -N x_{ij} \leq \pi_j - \pi_i \leq N x_{ij} \\
 & \text{(no edge from i to j, but there is a fusion benefit)} \\
 & x_{ij} \quad \leq \pi_j - \pi_i \leq N x_{ij} \\
 & \text{(an edge from i to j)} \\
 & \quad \pi_i < \pi_j \\
 & \text{(a } \textit{fusion-preventing} \text{ edge from i to j)} \\
 & \\
 & x_{ij} \quad \leq c_i \\
 & \text{(for all edges from i with no fusion-preventing outputs)} \\
 & \\
 & x_{i'j} \quad \leq x_{ij} \\
 & \text{(if } \tau_i \neq \tau_j \wedge gen_i = \{i'\}) \\
 & x_{ij'} \quad \leq x_{ij} \\
 & \text{(if } \tau_i \neq \tau_j \wedge gen_j = \{j'\})
\end{array}
$$

4.4 Proof

To prove correctness of our linear program formulation, we need to prove two different things. Firstly, the formulation's constraints must always be satisfiable; that is, there must exist a variable assignment that satisfies all constraints. This is rather simple to show, but guarantees that the linear program will always give an answer. The next thing to show is that any produced clustering is legal: if a variable assignment satisfies the constraints, then it is a valid and legal clustering. This means that, not only do we get *an* answer, we also get the *right* answer.

4.4.1 Satisfiability

For any program p, there exists a trivial clustering with no fusion at all. We can use this as the variable assignment of $ilp(p)$. For each pair of nodes $m, n \in p$, $x_{mn} = 1$ — no fusion is possible. For the π variables, we must find a topographical ordering of the nodes in p, which is simple since we are assured it is a dag.

TODO: *Now, prove that this assignment actually satisfies the constraints.*

4.4.2 Soundness

For any program p and variable asignment v, if v satisfies the constraints for $ilp(p)$, the clustering denoted by x_{ij} in v is legal.

For a clustering to be legal, it must satisfy three constraints:

Acyclic after merging nodes of same cluster together, the resulting graph must be a dag

Precedence preserving if there is an edge between two nodes i and j, and they are not merged together, then we require $\pi_j > \pi_i$

Fusion preventing likewise, if there is a fusion-preventing edge between two nodes i and j, then we require $\pi_j > \pi_i$, which implies that they are not merged together

Type constraint if two nodes i and j are in the same cluster, then $\tau_i = \tau_j$, or if τ_i is a subtype of τ_j (or τ_j is a subtype of τ_i), then the *generator* for τ_i (or τ_j) must also be in the same cluster as i and j.

TODO: Actually, let us say $x_{ij} = 0 \implies check_{ij}$

where

$$
\begin{array}{lll}
check :: array \times array \to \mathbb{B} & & \\
check(i,j) \mid tau_i = tau_j & = & x_{i,j} = 0 \\
check(i,j) \mid i' \in gen(i) & = & x_{i',j} = 0 \wedge x_{i,i'} = 0 \wedge check(i', j) \\
check(i,j) \mid j' \in gen(j) & = & x_{i,j'} = 0 \wedge x_{j,j'} = 0 \wedge check(i, j') \\
check(i,j) \mid tau_i \neq tau_j & = & \bot
\end{array}
$$

	Unfused		Stream		Megiddo		**Ours**	
	Time	Loops	Time	Loops	Time	Loops	Time	Loops
Normalise2	1.88s	5	1.64s	4	1.82s	3	**1.59s**	**2**
Closest pair	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 6. Benchmark results

5. Benchmarks

BL: *Use the larger programs as benchmarks, or as running examples. We only need small, simple programs to demonstrate how the algorithm works.*

Notes:

- In some cases, the clusterings were the same for different methods. In this case, the results are the same.
- The stream fusion benchmarks use the clustering that stream fusion *would* use, but hand-fused and written in C.
- Programs were run five times with the same input data, and the fastest run was used.
- ILP solutions were created by hand-fusing based on the clustering of the implementation. In the future, this will be integrated to use data flow fusion.
- Benchmark programs are available at `https://github.com/amosr/papers/tree/master/2014betterfusionforfilters/benches`

5.1 Closest pairs

Closest pairs makes use of fast-ish `median` to ensure balanced division of work.

```
closest :: Vec Pt -> (Pt,Pt)
closest pts
 | length pts < 250
 = naive pts
 | otherwise
 = divide pts

divide :: Vec Pt -> (Pt,Pt)
divide pts
 = let p      = median pts
       aboves = filter (above p) pts
       belows = filter (below p) pts
       above' = closest aboves
       below' = closest belows

       border = min (distance above') (distance below')

       aboveB = filter (above (p - border)) pts
       belowB = filter (below (p + border)) pts

       cs     = cross aboveB belowB
       bord   = minBy distance cs
   in  above' `minDist` below' `minDist` bord


naive :: Vec Pt -> (Pt,Pt)
naive pts
 = let c = cross pts pts
   in  minBy distance c

minBy :: Ord b => (a -> b) -> Vec a -> a
minBy f xs
 = fold (min . f...) ... xs
```

Let's translate `divide` to a program in CNF.

```
divide :: Vec Pt -> (Pt,Pt)
divide pts
 = let p      = external pts

       aboves = filter (... p) pts          -- A
       belows = filter (... p) pts          -- A

       above' = external aboves
       below' = external belows
       border = external above' below'

       aboveB = filter (... p border) pts   -- C
       belowB = filter (... p border) pts   -- B

       cs     = cross  aboveB belowB        -- C
       bord   = fold   (...) cs             -- C

       min'   = external above' below' bord
   in  min'
```

where `A`, `B` and `C` are distinct clusters.

What happens if we do this using stream fusion?

```
divide :: Vec Pt -> (Pt,Pt)
divide pts
 = let p      = external pts

       aboves = filter (... p) pts          -- A
       belows = filter (... p) pts          -- B

       above' = external aboves
       below' = external belows
       border = external above' below'


       aboveB = filter (... p border) pts   -- D
       belowB = filter (... p border) pts   -- C

       cs     = cross  aboveB belowB        -- D
       bord   = fold   (...) cs             -- D

       min'   = external above' below' bord
   in  min'
```

So, we have four clusters instead of three, and the same number of manifest arrays. Not particularly impressive.

5.2 Quickhull

Is Quickhull any better? Seems like it's just one cluster; `filterMax`. And we don't have append (`++`), anyway.

```
quickhull :: Vec Pt -> Vec Pt
quickhull pts
 = let top  = fold getTop pts
       bot  = fold getBot pts
```

```
      tops = hull (top,bot) pts
      bots = hull (bot,top) pts
  in  tops ++ bots

hull :: (Pt,Pt) -> Vec Pt -> Vec Pt
hull line@(l,r) pts
 = let pts' = filter (above   line) pts
       ma   = fold   (maxFrom line) pts'
       hl   = hull   (l, ma)        as
       hr   = hull   (ma, r)        as
  in  hl  ++ hr
```

Yep. `hull` is pretty boring.

Something with a *fold*, and then filtering or mapping based on that fold would be good. Or something with

```
let xs' = filter f    xs
    i   = fold   g    xs'
    xs''= map   (h i) xs'
```

would be really good.

FFT - not really.

5.3 QuadTree

```
bounds pts
 = let x1 = fold minX pts
       y1 = fold minY pts
       x2 = fold maxX pts
       y2 = fold maxY pts
   in ((x1,y1), (x2,y2))

%splitbox ((x1,y1), (x2,y2))
% = let xm = mid x1 x2
%       ym = mid y1 y2
%   in ( ((x1, y1), (xm, ym))
%      , ((x1, ym), (xm, y2))
%      , ((xm, y1), (x2, ym))
%      , ((xm, ym), (x2, y2)))
%
quadtree pts
 = go (bounds pts)
 where
  go bounds pts
   | length pts > 0
   = let (b1,b2,b3,b4) = splitbox bounds
         pts1  = filter (inbox b1) pts
         pts2  = filter (inbox b2) pts
         pts3  = filter (inbox b3) pts
         pts4  = filter (inbox b4) pts
         tree1 = go b1 pts1
         tree2 = go b2 pts2
         tree3 = go b3 pts3
         tree4 = go b4 pts4
     in Tree tree1 tree2 tree3 tree4
   | otherwise
   = Empty
```

It's easy to see that `bounds` should only require one loop, but stream fusion requires four, as there is no inlining that can occur. The same is true of `go` in `quadtree`. Our implementation only requires one loop for each of these.

6. Related work

TODO: *Cite work on clock inference for data flow languages. Eg http://www.irisa.fr/prive/talpin/papers/hldvt05.pdf*

6.1 Haskell short-cut fusion

Existing fusion systems for Haskell such as stream fusion[**?** **?**], tend to cleverly reuse compiler optimisations such as inlining and rewrite rules[**?**], to fuse combinators without having to modify the compiler itself. This approach has the advantage of simplicity, but is inherently limited in the amount of fusion it can perform.

Consider the following `filterMax` function, where each element in the input array is incremented, then the maximum is found, and the array is filtered to those greater than zero. In this case, the result of the map `vs'` cannot be inlined into both occurences without duplicating work, so no fusion can be performed. This has the effect of performing three loops instead of one, with two arrays instead of one.

```
filterMax vs =
 let vs' = map    (+1)  vs
     m   = fold   max 0 vs'
     flt = filter (>0)  vs'
```

6.2 Integer linear programming in imperative languages

The idea of using integer linear programming to find optimal fusion clusterings is not new, and has been discussed for imperative languages before. These methods first construct a *loop dependence graph* (LDG) from a given program, and then use this graph to create the integer linear program. The LDG has nodes for each loop in the program, and edges between loops are dependencies. Edges may be fusible, or fusion-preventing, in which case the two nodes may not be merged together.

Talk about the simple formulation by Darte[**?**]. Has an integer variable for each node, denoting the number of the cluster it's in. Also includes a binary variable for each node, which is whether the node is fused with all its successors — in which case no array would be required, and an integer variable which is the maximum of all cluster numbers. The objective function is to maximise the number of nodes that are completely fused, requiring no arrays, and minimise the maximum cluster number, which in turn minimises the number of clusters. It doesn't require many constraints and is easy to implement, but doesn't work as well when there are loops of different sizes. As loops of different sizes cannot be fused together, a simple method is to introduce an ordering on the sizes, and then extract loops of the same cluster number in order of size. The problem here is that the objective function uses the maximum cluster number to minimise the number of loops, but this alone is no longer sufficient when there are multiple sizes. ***TODO:*** *Explain why. Perhaps go into more detail about how arrays are contracted, which is different from Megiddo.*

The formulation by Megiddo[**?**] supports different sized loops, and is therefore more relevant for our purposes. For every pair of nodes i, j in the LDG, a variable x_{ij} is created, which denotes whether i and j are fused together. Slightly awkwardly, but for simplicity of other constraints, $x_{ij} = 0$ if the two nodes are fused together. If there is a fusion preventing edge between i and j, then x_{ij} is constrained to be 1 — that is, no fusion is possible. This alone is not enough to guarantee a valid clustering. To constrain the solution to acyclic and precedence preserving clusterings, a variable π_i is added for each node i. Constraints are added that require two nodes i, j to have $\pi_i = \pi_j$ if $x_{ij} = 0$, and otherwise $\pi_i > \pi_j$ if i is after j. For each pair of nodes, a weight constant w_{ij} is given, and the objective function is to minimise $\Sigma_{ij} w_{ij} x_{ij}$, which has the effect of maximally fusing nodes, according to their weights.

The difference to our combinator-based approach is that with combinators we retain more information about the meaning of the program.

6.3 Non-optimal heuristics

[**?**] Collective loop fusion for array contraction. I think this is a good introduction to the min/max edge following algorithm that is central to most of these. Finds minimal number of clusters for single types. Despite the name, I don't think it actually minimises clustering for array contraction.

[**?**] Ken Kennedy — Typed Fusion with Applications to Parallel and Sequential Code Generation. Typed fusion — choose an ordering of types, then find clustering of first type, fuse those together, then cluster second type, and so on. It ain't optimal, even if you try all orderings (see Darte below)

[**?**] Chatterjee – Not optimal, heuristic based, etc, but does actually use ILP to reduce array crossing clusterings. Like typed fusion, they choose clustering for different types rather arbitrarily, which is sub-par.

[**?**] Alain Darte — On the complexity of loop fusion. Shows that loop fusion, minimising certain things (ie manifest arrays and number of loops) is NP-hard.

[**?**] Improving data locality by array contraction. All about *controlled SFC*, ie shifting, fusion and contraction. Formalises cost of memory references depending on distance / how many other references in between. Then only fuses loops if it doesn't raise the distance too much.

[**?**] T.J. Ashby — Iterative collective loop fusion. Executes the programs to decide which clustering is best, apparently. I didn't read it thoroughly, but didn't understand where they actually get the test data from. Unless it's a kind of JIT thing. I honestly don't think much of it, but it *is* more recent than the other stuff, so I suspect it'd be good to mention just so we're not ignoring "modern fusion".

[**?**] Ken Kennedy — Fast Greedy Weighted Fusion.

[**?**] Optimization of array accesses by collective loop transformations. This is probably not at all relevant, but it's *very cute*. Use a two-colouring algorithm to decide when to reverse loops, to get best fusion. Not our scene.

The polyhedral model is an algebraic representation of imperative loop nests and transformations on them, including fusion transformations. Polyhedral systems [**?**] 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 dimentional 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 [**?**], as well as conditional control flow that is predicated on arbitrary (ie, non-affine) functions of the loop indices [**?**]. However, the indices used to write into the destination array must still be computed with affine functions.

TODO: *Is unimodular the same as the polyhedral model?. Now, here's my understanding of unimodular (and I'm just guessing this applies to polyhedral too): if the dependency matrix for a loop nest or set of loops forms a* unimodular matrix *(an integral matrix whose inverse is also integral), only then can it be dealt with by unimodular transforms.*

7. Conclusion

Acknowledgements

Many thanks are due to Robert Clifton-Everest, Kai Engelhardt, Bill Kroon, Frederik Madsen, Abdallah Saffidine, Carter Schonwald, and Jingling Xue for enlightening discussions relating to this work.