Machine Fusion

Merging merges, more or less

Amos Robinson
Ambiata and UNSW (Australia)
amosr@cse.unsw.edu.au

Ben Lippmeier
Digital Asset and UNSW (Australia)
benl@ouroborus.net

ABSTRACT

Compilers for stream programs often rely on a fusion transformation to convert the implied dataflow network into low-level iteration based code. Different fusion transformations handle different sorts of networks, with the distinguishing criteria being whether the network may contain splits and joins, and whether the set of fusible operators can be extended. We present the first fusion system that simultaneously satisfies all three of these criteria: networks can contain splits, joins, and new operators can be added to the system without needing to modify the overall fusion transformation.

CCS CONCEPTS

• **Theory of computation** → **Streaming models**;

1 INTRODUCTION

Suppose we have two input streams of numeric identifiers, and wish to perform some analysis on them. The identifiers from both streams are sorted, but may include duplicates. We wish to produce an output stream of unique identifiers from the first input stream, as well as produce the unique union of both streams. Here is how we might write the source code, where S is for S-tream.

```
uniquesUnion : S Nat -> S Nat -> (S Nat, S Nat)
uniquesUnion sIn1 sIn2
 = let  sUnique = group sIn1
        sMerged = merge sIn1 sIn2
        sUnion  = group sMerged
   in   (sUnique, sUnion)
```

The `group` operator detects groups of consecutive identical elements and emits a single representative, while `merge` combines two sorted streams so that the output remains sorted. This example has a few interesting properties. Firstly, the data-access pattern of `merge` is *value-dependent*, meaning that the order in which `merge` pulls values from `sIn1` and `sIn2` depends on the values themselves.

Secondly, although `sIn1` occurs twice in the program, at runtime we only want to handle its elements once. To achieve this, the compiled program must coordinate between the two uses of `sIn1`, so that a new value is read only when both `group` and `merge` are ready to receive it. Finally, as the stream length is unbounded, we cannot buffer arbitrarily many elements, or we risk running out of memory.

To implement this we might translate each operator into its own process in a concurrent network. This could be easy or hard, depending on the available language features for concurrency. However, the *performance tuning* of such a system, such as using back-pressure to prevent buffers from being overrun, or how to chunk stream data to amortize communication overhead, is invariably a headache.

Instead, we would prefer to use *stream fusion*, which is a program transformation that takes the implied dataflow network and produces a simple sequential loop that does not require extra process-control abstractions or unbounded buffering. Sadly, existing stream fusion transformations cannot handle our example.

As observed by Kay [14], both pull and push fusion have fundamental limitations. Pull systems such as short-cut stream fusion [9] cannot handle cases where a stream is used by multiple consumers. We refer to this situation as a *split* — in our example the input stream `sIn1` is split into both the `group` and `merge` consumers. Push systems such as foldr/build fusion [10] cannot fuse our example either, because they do not support operators with multiple inputs. We refer to this as a *join* — in our example the `merge` operator expresses a join. Some systems support both: data flow inspired array fusion using series expressions [19] allows both splits and joins but only for a limited, predefined set of operators. More recent work on polarized data flow fusion [21] *is* able to fuse our example, but requires the program to be rewritten to use explicitly polarized stream types.

In this paper we present Machine Fusion, a new approach. Each operator is expressed as a sequential imperative program which *pulls* from input streams, and *pushes* to output streams. We view each operator as a process in a concurrent process network. Our fusion transform then *sequentializes* the concurrent process network into a single process, by choosing a particular interleaving of the operator code that requires no unbounded intermediate buffers. When fusion succeeds we know it has worked. There is no need to inspect the compiled code to debug poor performance, which is a common problem in systems based on general purpose transformations [20].

In summary, we make the following contributions:

- a process calculus for infinite streaming programs (§2);
- a fusion algorithm, the first to support splits and joins (§4);
- benchmarks showing significant performance gains (§5);
- proof of soundness for the fusion algorithm in Coq (§6).

Our fusion transform for infinite stream programs also serves as the basis for an *array* fusion system, using a natural extension to finite streams. We discuss this extension in §5.1.

2 PROCESSES AND MACHINES

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

A *combinator* is a template for a process which parameterizes it over the particular input and output streams, as well as values of configuration parameters such as the worker function used in a map process. Each process implements a logical *operator* — so we use "operator" when describing the values being computed, but "process" and "machine" when referring to the implementation.

2.1 Grouping

The definition of the group combinator which detects groups of successive identical elements in the input stream is given in Figure 1. The process emits the first value pulled from the stream and every value that is different from the last one that was pulled. For example, when executed on the input stream [1,2,2,3], the process will produce the output [1,2,3]. We include the concrete representation and a diagram of the process when viewed as a state machine.

The group combinator has two parameters, sIn1 and sOut1, which bind the input and output streams respectively. The *nu-binders* (ν (f: Bool) (l: Nat)...) indicate that each time the group combinator is instantiated, fresh names must be given to f, l and so on, that do not conflict with other instantiations. Overall, the f variable tracks whether we are dealing with the first value from the stream, l holds the last value pulled from the stream (or 0 if none have been read yet), and v holds the current value pulled from the stream.

The body of the combinator is a record that defines the process. The ins field defines the set of input streams and the outs field the set of output streams. The heap field gives the initial values of each of the local variables. The instrs field contains a set of labeled instructions that define the program, while the label field gives the label of the initial instruction. In this form, the output stream is defined via a parameter, rather than being the result of the combinator, as in the representation of uniquesUnion from §1.

The initial instruction (pull sIn1 v A1 []) pulls the next element from the stream sIn1, writes it into the heap variable v (value), then proceeds to the instruction at label A1. The empty list [] after the target label A1 can be used to update heap variables, but as we do not need to update anything yet we leave it empty.

Next, the instruction (case (f || (l /= v)) A2 [] A3 []) checks whether predicate (f || (l /= v)) is true; if so it proceeds to the instruction at label A2, otherwise it proceeds to A3. We use the variable l (last) to track the last value read from the stream, and the boolean f (first) to track whether this is the first element.

When the predicate is true, the instruction at label A2 executes (push sOut1 v A3 [l = v, f = F]) which pushes the value v to the output stream sOut1 and proceeds to the instruction at label A3, once the variable l is set to v and f to F (False).

Finally, the instruction (drop sIn1 A0 []) signals that the current element that was pulled from stream sIn1 is no longer required, and goes back to the first instruction at A0.

2.2 Merging

The definition of the merge combinator, which merges two input streams, is given in Figure 2. The combinator binds the two input streams to sIn1 and sIn2, while the output stream is sOut2. The two heap variables x1 and x2 store the last values read from each input stream. The process starts by pulling from each of the input streams. It then compares the two pulled values, and pushes the smaller of the values to the output stream. The process then drops the stream which yielded the the smaller value, then pulls from the same stream so that it can perform the comparison again.

2.3 Fusion

Our fusion algorithm takes two processes and produces a new one that computes the output of both. For example, suppose we need a single process that produces the output of the first two lines of our uniquesUnion example back in §1. The result will be a process that computes the result of both group and merge as if they were executed concurrently, where the first input stream of the merge process is the same as the input stream of the group process. In our informal description of the fusion algorithm we will instantiate the parameters of each combinator with arguments of the same names.

2.3.1 Fusing Pulls. The algorithm proceeds by considering pairs of states: one from each of the source process state machines to be fused. Both the group machine and the merge machine pull from the same stream as their initial instruction, so we have the situation shown in the top of Figure 3. The group machine needs to transition from label A0 to label A1, and the merge machine from B0 to B1. In the result machine we produce three new instructions that transition between four joint result states, F0 to F3. Each of the joint result states represents a combination of two source states, one from each of the source machines. For example, the first result state F0 represents a combination of the group machine being in its initial state A0 and the merge machine being in its own initial state B0.

We also associate each of the joint result states with a description of whether each source machine has already pulled a value from each of its input streams. For the F0 case at the top of Figure 3 we have ((A0, {sIn1 = none}), (B0, {sIn1 = none, sIn2 = none})). The result state F0 represents a combination of the two source states A0 and B0. As both A0 and B0 are the initial states of their respective machines, those machines have not yet pulled any values from their two input streams, so both 'sIn1' and 'sIn2' map to 'none'.

From the result state F0, both of the source machines then need to pull from stream sIn1, the group machine storing the value in a variable v and the merge machine storing it in x1. In the result machine this is managed by first storing the pulled value in a fresh, shared buffer variable b1, and then using later instructions to copy the value into the original variables v and x1. To perform the copies we attach updates to a jump instruction, which otherwise transitions between states without affecting any of the input or output streams.

Finally, note that in the result states F0 through F3, the state of the input streams transitions from 'none', to 'pending' then to 'have'. The 'none' state means that we have not yet pulled a value from the associated stream. The 'pending' state means we have pulled a value into the stream buffer variable (b1 in this case). The 'have' state means that we have copied the pulled value from the stream buffer variable into the local variable used by each machine.

```
group
  = λ (sIn1: Stream Nat) (sOut1: Stream Nat).
    ν (f: Bool) (l: Nat) (v: Nat) (A0..A3: Label).
    process
    { ins:    { sIn1  }
    , outs:   { sOut1 }
    , heap:   { f = T, l = 0, v = 0 }
    , label:  A0
    , instrs: { A0 = pull sIn1 v           A1 []
              , A1 = case (f || (l /= v))  A2 []   A3 []
              , A2 = push sOut1 v          A3 [ l = v, f = F ]
              , A3 = drop sIn1             A0 [] } }
```
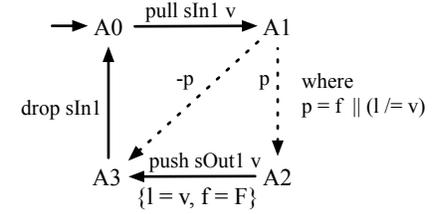

Figure 1: The group combinator

```
merge
  = λ (sIn1: Stream Nat) (sIn2: Stream Nat) (sOut2: Stream Nat).
    ν (x1: Nat) (x2: Nat) (B0..E2: Label).
    process
    { ins:    { sIn1, sIn2 }
    , outs:   { sOut2 }
    , heap:   { x1 = 0, x2 = 0 }
    , label:  B0
    , instrs: { B0 = pull sIn1  x1   B1 []          , B1 = pull sIn2  x2   C0 []
              , C0 = case (x1 < x2)  D0 []  E0 []   , D0 = push sOut2 x1   D1 []
              , D1 = drop sIn1       D2 []          , D2 = pull sIn1  x1   C0 []
              , E0 = push sOut2 x2   E1 []          , E1 = drop sIn2       E2 []
              , E2 = pull sIn2 x2    C0 [] } }
```
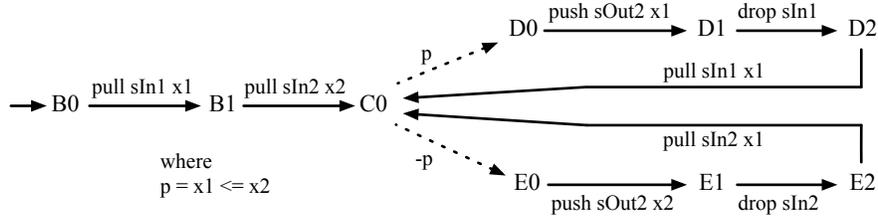

Figure 2: The merge combinator

2.3.2 Fusing Cases. Once the result machine has arrived at the joint state F3, this is equivalent to the two source machines arriving in states A1 and B1 respectively. The lower half of Figure 3 shows the next few transitions of the source machines. From state A1, the group machine needs to perform a case branch to determine whether to push the current value it has from its input stream sIn1 to output stream sOut1, or to just pull the next value from its input. From state B1, the merge machine needs to pull a value from its second input stream sIn2. In the result machine, F3 performs the case analysis from A1, moving to either F4 or F5, corresponding to A2 and A3 respectively. From state F4, the push at A2 is executed and moves to F5, corresponding to A3. Finally, at F5 the merge machine pulls from sIn2, moving from F5 to F6, corresponding to B1 and C0 respectively. As the stream sIn2 is only pulled from by the merge machine, no coordination is required between the merge and group machines for this pull.

2.4 Fused Result

Figure 4 shows the final result of fusing group and merge together. There are similar rules for handling the other combinations of instructions, but we defer the details to §4. The result process has two input streams, sIn1 and sIn2, and two output streams: sOut1 from group, and sOut2 from merge. The shared input sIn1 is pulled by merge instructions at two places, and since both of these need to agree with when group pulls, the group instructions are duplicated at F3-F5 and F13-F15. The first set of instructions could be simplified by constant propagation to a single push, as f is initially true.

To complete the implementation of our example from §1 we would now fuse this result process with a process from the final line of the example (also a group). Note that although the result process has a single shared heap, the heap bindings from each fused process are guaranteed not to interfere, as when we instantiate combinators to create source processes we introduce fresh names.

2.5 Breaking It Down

We started with a pure functional program in §1, reimagined it as a dataflow graph, then interleaved imperative code that implemented two of the operators in that dataflow graph. We needed to *break down* the definition of each operator into imperative statements so that we could interleave their execution appropriately. We do this because the standard, single-threaded evaluation semantics of functional programs does not allow us to evaluate stream programs that contain both splits and joins in a space efficient way. Returning to the definition of uniquesUnion from §1, we cannot simply execute the group operator on its entire input sIn1 before executing the merge operator, as that would require us to buffer all data read from sIn1. Instead, during fusion we perform the job of a concurrent scheduler at compile time. In the result process the flow of control alternates between the instructions for both the group and merge operators, but as the instructions are interleaved directly there is no overhead due to context switching — as there would be in a standard concurrent implementation using multiple threads.

The general approach of converting a pure functional program to a dataflow graph, then interleaving imperative statements that implement each operator was also used in prior work on Flow Fusion [19]. However, in contrast to Flow Fusion and similar systems, with Machine Fusion we do not need to organize statements into a fixed *loop anatomy* — we simply merge them as they are. This allows us to implement a wider range of operators, including ones with nested loops that work on segmented streams.

Note that relying on lazy evaluation for uniquesUnion does not eliminate the need for unbounded buffering. Suppose we converted each of the streams to lazy lists, and used definitions of group and merge that worked over these lists. As uniquesUnion returns a pair of results, there is nothing preventing a consumer from demanding the first list (sUnique) in its entirety before demanding any of the elements from the second list (sUnion). If this were to happen then the runtime implementation would need to retain all elements of sIn1 before demanding any of sIn2, causing a space leak. Lazy evaluation is *pull only* meaning that evaluation is driven by the consumer. The space efficiency of our fused program relies critically on the fact that processes can also *push* their result values directly to their consumers, and that the consumers cannot defer the handling of these values.

3 PROCESS DEFINITIONS

The formal grammar for process definitions is given in Figure 5. Variables, Channels and Labels are specified by unique names. We refer to the *endpoint* of a stream as a channel. A particular stream may flow into the input channels of several different processes, but can only be produced by a single output channel. For values and expressions we use an untyped lambda calculus with a few primitives chosen to facilitate the examples. The '||' operator is boolean-or, '+' addition, '/=' not-equal, and '<' less-than.

A *Process* is a record with five fields: the ins field specifies the input channels; the outs field the output channels; the heap field the process-local heap; the label field the label of the instruction currently being executed, and the instrs a map of labels to instructions. We use the same record when specifying both the definition of a particular process, as well as when giving the evaluation semantics.

When specifying a process the label field gives the entry-point to the process code, though during evaluation it is the label of the instruction currently being executed. Likewise, when specifying a process we usually only list channel names in the ins field, though during evaluation they are also paired with their current *InputState*. If an *InputState* is not specified we assume it is 'none'. A network is a set of processes that are able to communicate with each other.

In the grammar of Figure 5 the *InputState* has three options: none, which means no value is currently stored in the associated stream buffer variable, (pending *Value*) which gives the current value in the stream buffer variable and indicates that it has not yet been copied into a process-local variable, and have which means the pending value has been copied into a process-local variable. The *Value* attached to the pending state is used when specifying the evaluation semantics of processes. When performing the fusion transform the *Value* itself will not be known, but we can still reason statically that a process must be in the pending state. When defining the fusion transform in §4 we will use a version of *InputState* with only this statically known information.

The instrs field of the *Process* maps labels to instructions. The possible instructions are: pull, which pulls the next value from a channel into a given heap variable; push, which pushes the value of an expression to an output channel; drop which indicates that the current value pulled from a channel is no longer needed; case which branches based on the result of a boolean expression, and jump which causes control to move to a new instruction.

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

When lowering process code to a target language, such as C, LLVM, or some sort of assembly code, we can safely convert drop to plain jump instructions. The drop instructions are used to control how processes should be synchronized, but do not affect the execution of a single process. We discuss drops further in §5.3.

3.1 Execution

The dynamic execution of a process network consists of:

(1) *Injection* of a single value from a stream into a process, or a network. Each individual process only needs to accept an injected value when it is ready for it, and injection into a network succeeds only when they *all* processes accept it.

(2) *Advancing* a single process from one state to another. Advancing a network succeeds when *any* of the processes in the network can advance.

(3) *Feeding* outputs of some processes to the inputs of others. Feeding alternates between Injecting and Advancing. When a process pushes a value to an output channel we attempt to inject this value into all processes that have that same channel as an input. If they all accept it, we then advance their programs as far as they will go, which may cause more values to be pushed to output channels, and so on.

Execution of a network is non-deterministic. At any moment several processes may be able to take a step, while others are blocked. As with Kahn processes [13], pulling from a channel is blocking,
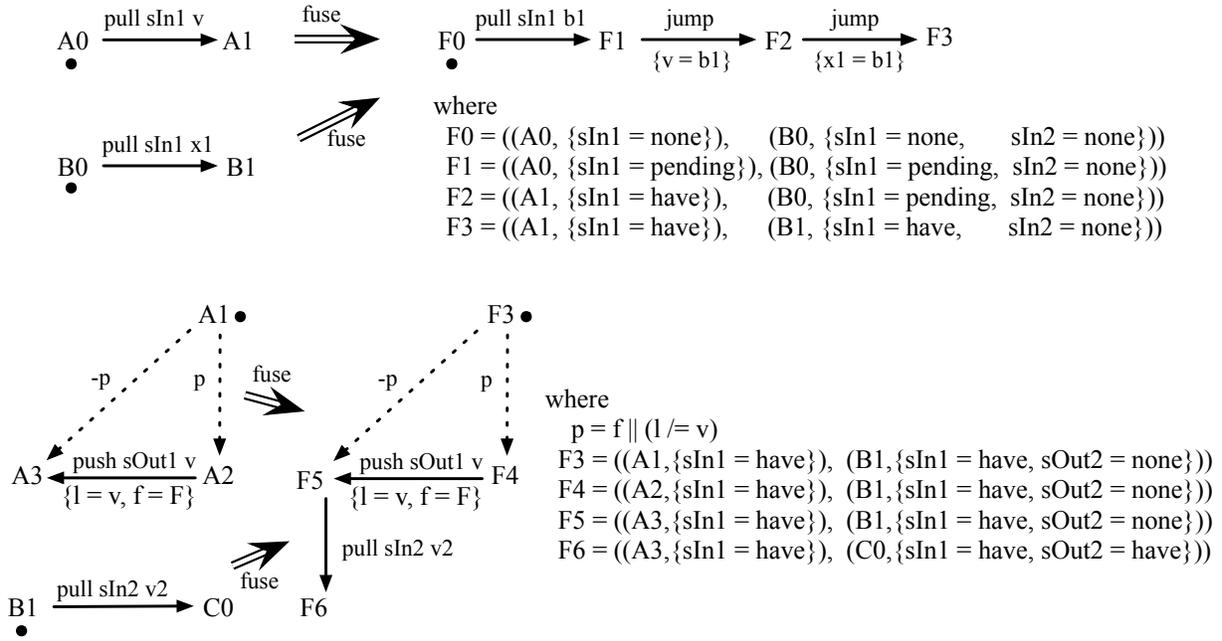
$$A0 \xrightarrow{\text{pull sIn1 v}} A1 \qquad \xRightarrow{\text{fuse}} \qquad F0 \xrightarrow{\text{pull sIn1 b1}} F1 \xrightarrow[\{v = b1\}]{\text{jump}} F2 \xrightarrow[\{x1 = b1\}]{\text{jump}} F3$$

$$B0 \xrightarrow{\text{pull sIn1 x1}} B1 \qquad \xRightarrow{\text{fuse}}$$

where

F0 = ((A0, {sIn1 = none}),   (B0, {sIn1 = none,   sIn2 = none}))
F1 = ((A0, {sIn1 = pending}), (B0, {sIn1 = pending, sIn2 = none}))
F2 = ((A1, {sIn1 = have}),   (B0, {sIn1 = pending, sIn2 = none}))
F3 = ((A1, {sIn1 = have}),   (B1, {sIn1 = have,   sIn2 = none}))

where

p = f || (l /= v)
F3 = ((A1,{sIn1 = have}), (B1,{sIn1 = have, sOut2 = none}))
F4 = ((A2,{sIn1 = have}), (B1,{sIn1 = have, sOut2 = none}))
F5 = ((A3,{sIn1 = have}), (B1,{sIn1 = have, sOut2 = none}))
F6 = ((A3,{sIn1 = have}), (C0,{sIn1 = have, sOut2 = have}))

Figure 3: Fusing pull (top) and case (bottom) instructions

```
process
{ ins:   { sIn1,  sIn2 }
, outs:  { sOut1, sOut2 }
, heap:  { f = T, l = 0, v = 0, x1 = 0, x2 = 0, b1 = 0 }
, label:  F0
, instrs:
  { F0  = pull sIn1 b1       F1 [ ]              F0  = ((A0,{sIn1 = none}),   (B0, {sIn1 = none,    sIn2 = none}))
  , F1  = jump               F2 [ v = b1 ]       F1  = ((A0,{sIn1 = pending}),(B0, {sIn1 = pending, sIn2 = none}))
  , F2  = jump               F3 [ x1 = b1 ]      F2  = ((A1,{sIn1 = have}),   (B0, {sIn1 = pending, sIn2 = none}))
  , F3  = case (f || (l /= v)) F4 [ ]    F5 [ ]  F3  = ((A1,{sIn1 = have}),   (B1, {sIn1 = have,    sIn2 = none}))
  , F4  = push sOut1 v       F5 [ l = v, f = F ] F4  = ((A2,{sIn1 = have}),   (B1, {sIn1 = have,    sIn2 = none}))
  , F5  = jump               F6 [ ]              F5  = ((A3,{sIn1 = have}),   (B1, {sIn1 = have,    sIn2 = none}))
  , F6  = pull sIn2 x2       F7 [ ]              F6  = ((A0,{sIn1 = none}),   (B1, {sIn1 = have,    sIn2 = none}))

  , F7  = case (x1 < x2)     F8 [ ]    F16 [ ]   F7  = ((A0,{sIn1 = none}),   (C0, {sIn1 = have,    sIn2 = have}))

  , F8  = push sOut2 x1      F9 [ ]              F8  = ((A0,{sIn1 = none}),   (D0, {sIn1 = have,    sIn2 = have}))
  , F9  = drop sIn1          F10 [ ]             F9  = ((A0,{sIn1 = none}),   (D1, {sIn1 = none,    sIn2 = have}))
  , F10 = pull sIn1 b1       F11 [ ]             F10 = ((A0,{sIn1 = none}),   (D2, {sIn1 = none,    sIn2 = have}))
  , F11 = jump               F12 [ v = b1 ]      F11 = ((A0,{sIn1 = pending}),(D2, {sIn1 = pending, sIn2 = have}))
  , F12 = jump               F13 [ x1 = b1 ]     F12 = ((A1,{sIn1 = have}),   (D2, {sIn1 = pending, sIn2 = have}))
  , F13 = case (f || (l /= v)) F14 [ ]   F15 [ ] F13 = ((A1,{sIn1 = have}),   (C0, {sIn1 = have,    sIn2 = have}))
  , F14 = push sOut1 v       F15 [ l = v, f = F ] F14 = ((A2,{sIn1 = have}),  (C0, {sIn1 = have,    sIn2 = have}))
  , F15 = jump               F7 [ ]              F15 = ((A3,{sIn1 = have}),   (C0, {sIn1 = have,    sIn2 = have}))

  , F16 = push sOut2 x2      F17 [ ]             F16 = ((A0,{sIn1 = none}),   (E0, {sIn1 = have,    sIn2 = have}))
  , F17 = drop sIn2          F18 [ ]             F17 = ((A0,{sIn1 = none}),   (E1, {sIn1 = have,    sIn2 = have}))
  , F18 = pull sIn2          F7 [ ]              F18 = ((A0,{sIn1 = none}),   (E2, {sIn1 = have,    sIn2 = none}))
  } }
```

Figure 4: Fusion of group and merge, along with shared instructions

$$
\begin{array}{ll}
Exp,\, e & ::= x \mid v \mid e\; e \\
& \mid\; (e \parallel e) \mid e + e \mid e \;/=\; e \mid e < e \\
Value,\, v & ::= \mathbb{N} \mid \mathbb{B} \mid (\lambda x.\; e) \\
Heap,\, bs & ::= \cdot \mid bs,\, x = v \\
Updates,\, us & ::= \cdot \mid us,\, x = e \\
\\
Process,\, p & ::= \texttt{process} \\
& \quad \texttt{ins:}\quad (Channel \mapsto InputState) \\
& \quad \texttt{outs:}\quad \{Channel\} \\
& \quad \texttt{heap:}\quad Heap \\
& \quad \texttt{label:}\quad Label \\
& \quad \texttt{instrs:}\; (Label \mapsto Instruction) \\
\\
InputState & ::= \texttt{none} \mid \texttt{pending}\; Value \mid \texttt{have}
\end{array}
$$

$$
\begin{array}{ll}
Variable,\, x & \rightarrow \text{(value variable)} \\
Channel,\, c & \rightarrow \text{(channel name)} \\
Label,\, l & \rightarrow \text{(label name)} \\
ChannelStates = & (Channel \mapsto InputState) \\
Action,\, a & ::= \cdot \mid \texttt{push}\; Channel\; Value \\
\\
Instruction & ::= \texttt{pull}\; Channel\; Variable\; Next \\
& \mid\; \texttt{push}\; Channel\; Exp \qquad Next \\
& \mid\; \texttt{drop}\; Channel \qquad\qquad Next \\
& \mid\; \texttt{case}\; Exp \qquad Next \qquad Next \\
& \mid\; \texttt{jump} \qquad\qquad\qquad Next \\
\\
Next & = Label \times Updates
\end{array}
$$

Figure 5: Process definitions

which enables the overall sequence of values on each output channel to be deterministic. Unlike Kahn processes, pushing to a channel can also block. Each consumer has a single element buffer, and pushing only succeeds when that buffer is empty.

Importantly, it is the order in which values are *pushed to each particular output channel* which is deterministic, whereas the order in which different processes execute their instructions is not. When we fuse two processes we choose one particular instruction ordering that enables the network to advance without requiring unbounded buffering. The single ordering is chosen by heuristically deciding which pair of states to merge during fusion, and is discussed in §3.2.

Each channel may be pushed to by a single process only, so in a sense each output channel is owned by a single process. The only intra-process communication is via channels and streams. Our model is "pure data flow" as there are no side-channels between processes — in contrast to "impure data flow" systems such as StreamIt [29].

3.1.1 Injection. Figure 6 gives the rules for injecting values into processes. Injection is a meta-level operation, in contrast to `pull` and `push` which are instructions in the object language. The statement $(p;\; \texttt{inject}\; v\; c\; \Rightarrow\; p')$ reads "given process p, injecting value v into channel c yields an updated process p'". The `injects` form is similar, operating on a process network.

Rule (InjectValue) injects a single value into a single process. The value is stored as a (`pending` v) binding in the *InputState* of the associated channel of the process. The *InputState* acts as a single element buffer, and must be empty (`none`) for injection to succeed.

Rule (InjectIgnore) allows processes that do not use a particular named channel to ignore values injected into that channel.

Rule (InjectMany) attempts to inject a single value into a network. We use the single process judgment form to inject the value into all processes, which must succeed for all of them.

3.1.2 Advancing. Figure 7 gives the rules for advancing a single process. The statement $(i;\; is;\; bs \overset{a}{\Rightarrow} l;\; is';\; us')$ reads "instruction i, given channel states is and the heap bindings bs, passes control to instruction at label l and yields new channel states is', heap update expressions us', and performs an output action a." An output action a is an optional message of the form (`push` *Channel Value*), which encodes the value a process pushes to one of its output channels. We write $\cdot$ for an empty action.

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

Rule (Push) evaluates the expression e under heap bindings bs to a value v, and produces a corresponding action which carries this value. The judgment $(bs \vdash e \Downarrow v)$ expresses standard untyped lambda calculus reduction using the heap bs for the values of free variables. As this evaluation is completely standard we omit it to save space.

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

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

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

3.1.3 Feeding. Figure 8 gives the rules for collecting output actions and feeding the contained values to other processes. The first set of rules concerns feeding values to other processes within the same network, while the second exchanges input and output values with the environment the network is running in.

The statement $ps \overset{a}{\Rightarrow} ps'$ reads "the network ps advances to the network ps' yielding output action a".

Rule (ProcessInternal) allows an arbitrary process in the network to advance to a new state at any time, provided it does not yield an output action. This allows processes to perform internal computation.

Rule (ProcessPush) allows an arbitrary process in the network to advance to a new state while yielding an output action (`push` $c\; v$). For this to succeed it must be possible to inject the output value v into all processes that have channel c as one of their inputs.

$$\boxed{Process; \; \texttt{inject} \; Value \; Channel \; \Rightarrow \; Process} \qquad \boxed{\{Process\}; \; \texttt{injects} \; Value \; Channel \; \Rightarrow \; \{Process\}}$$

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

$$\frac{c \notin p[\texttt{ins}]}{p; \; \texttt{inject} \; v \; c \; \Rightarrow \; p} \; \text{(InjectIgnore)} \qquad \frac{\{ \, p_i; \; \texttt{inject} \; v \; c \; \Rightarrow \; p_i' \, \}^i}{\{p_i\}^i; \; \texttt{injects} \; v \; c \; \Rightarrow \; \{p_i'\}^i} \; \text{(InjectMany)}$$

Figure 6: Injection of values into input channels

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

$$\frac{is[c] = \texttt{pending} \; v}{\texttt{pull} \; c \; x \; (l, us); \; is; \; bs \; \dot\Rightarrow \; l; \; is[c \mapsto \texttt{have}]; \; (us, x = v)} \; \text{(Pull)} \qquad \frac{bs \vdash e \Downarrow v}{\texttt{push} \; c \; e \; (l, us); \; is; \; bs \; \xRightarrow{\texttt{push} \; c \; v} \; l; \; is; \; us} \; \text{(Push)}$$

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

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

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

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

Figure 7: Advancing processes

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

$$\frac{p_i \; \dot\Rightarrow \; p_i'}{\{p_0 \ldots p_i \ldots p_n\} \; \dot\Rightarrow \; \{p_0 \ldots p_i' \ldots p_n\}} \; \text{(ProcessesInternal)} \qquad \frac{p_i \; \xRightarrow{\texttt{push} \; c \; v} \; p_i' \quad \forall j \mid j \neq i. \; p_j; \; \texttt{inject} \; c \; v \; \Rightarrow \; p_j'}{\{p_0 \ldots p_i \ldots p_n\} \; \xRightarrow{\texttt{push} \; c \; v} \; \{p_0' \ldots p_i' \ldots p_n'\}} \; \text{(ProcessesPush)}$$

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

$$\frac{ps \; \dot\Rightarrow \; ps'}{cvs; \; ps \; \Rightarrow \; cvs; \; ps'} \; \text{(FeedInternal)} \qquad \frac{ps \; \xRightarrow{\texttt{push} \; c \; v} \; ps'}{cvs; \; ps \; \Rightarrow \; cvs[c \mapsto (cvs[c] \; \texttt{++} \; v)]; \; ps'} \; \text{(FeedPush)}$$

$$\frac{(\forall p \in ps. \; c \notin p[\texttt{outs}]) \quad ps; \; \texttt{injects} \; c \; v \; \Rightarrow \; ps'}{cvs[c \mapsto ([v] \; \texttt{++} \; vs)]; \; ps \; \Rightarrow \; cvs[c \mapsto vs \,]; \; ps'} \; \text{(FeedExternal)}$$

Figure 8: Feeding Process Networks

The statement $cvs; \; ps \; \Rightarrow \; cvs'; \; ps'$ reads "with channel values cvs, network ps takes a step and produces new channel values cvs' and network ps'". The channel values cvs map channel names to a list of values. For input channels of the overall network, we initialize the map to contain a list of input values for each channel. For output channels of the overall network, values pushed to those channels are also collected in the same channel map. In a concrete implementation the input and output values would be transported over some IO device, but for the semantics we describe the abstract behavior only.

Rule (FeedInternal) allows the network to perform local computation in the context of the channel values.

Rule (FeedPush) collects an output action ($\texttt{push} \; c \; v$) produced by a network and appends the value v to the list corresponding to the output channel c.

Rule (FeedExternal) injects values from the external environment. This rule also has the side condition that values cannot be injected from the environment into output channels that are already owned by some process. This constraint is required for correctness proofs, but can be ensured by construction in a concrete implementation.

3.2 Non-deterministic Execution Order

The execution rules of Figure 8 are non-deterministic in several ways. Rule (ProcessInternal) allows any process to perform internal computation at any time, without synchronizing with other processes in the network; (ProcessPush) allows any process to perform a push action at any time, provided all other processes in the network are ready to accept the pushed value; (FeedExternal) also allows new values to be injected from the environment, provided all processes that use the channel are ready to accept the value.

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

4 FUSION

Our core fusion algorithm constructs a static execution schedule for a single pair of processes. To fuse a whole process network we fuse successive pairs of processes until only one remains.

Figure 9 defines some auxiliary grammar used during fusion. We extend the *Label* grammar with a new alternative, *LabelF* × *LabelF* for the labels in a fused result process. Each *LabelF* consists of a *Label* from a source process, paired with a map from *Channel* to the statically known part of that channel's current *InputState*. When fusing a whole network, as we fuse pairs of individual processes the labels in the result collect more and more information. Each label of the final, completely fused process encodes the joint state that all the original source processes would be in at that point.

We also extend the existing *Variable* grammar with a (chan *c*) form which represents the buffer variable associated with channel c. We only need one buffer variable for each channel, and naming them like this saves us from inventing fresh names in the definition of the fusion rules. We used a fresh name back in §2.3.1 to avoid introducing a new mechanism at that point in the discussion.

Still in Figure 9, *ChannelType2* classifies how channels are used, and possibly shared, between two processes. Type in2 indicates that the two processes pull from the same channel, so these actions must be coordinated. Type in1 indicates that only a single process pulls from the channel. Type in1out1 indicates that one process pushes to the channel and the other pulls. Type out1 indicates that the channel is pushed to by a single process. Each output channel is uniquely owned and cannot be pushed to by more than one process.

Figure 10 defines function *fusePair* that fuses a pair of processes, constructing a result process that does the job of both. We start with a joint label l_0 formed from the initial labels of the two source processes. We then use *tryStepPair* to statically choose which of the two processes to advance, and hence which instruction to execute next. The possible destination labels of that instruction (computed with *outlabels* from Figure 13) define new joint labels and reachable

$$
\begin{array}{ll}
\textit{Label} & ::= \dots \mid \textit{LabelF} \times \textit{LabelF} \mid \dots \\
\textit{LabelF} & = \textit{Label} \times (\textit{Channel} \mapsto \textit{InputStateF}) \\
\textit{InputStateF} & ::= \mathsf{none}_F \mid \mathsf{pending}_F \mid \mathsf{have}_F \\
\textit{Variable} & ::= \dots \mid \mathsf{chan}\ \textit{Channel} \mid \dots \\
\textit{ChannelType2} & ::= \mathsf{in2} \mid \mathsf{in1} \mid \mathsf{in1out1} \mid \mathsf{out1}
\end{array}
$$

Figure 9: Fusion type definitions.

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

Figure 10: Fusion of pairs of processes

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

Figure 11: Fusion step coordination for a pair of processes.

states. As we discover reachable states we add them to a map *bs* of joint label to the corresponding instruction, and repeat the process to a fixpoint where no new states can be discovered.

Figure 11 defines function *tryStepPair* which decides which process to advance. It starts by calling *tryStep* for both processes. If both can advance, we use heuristics to decide which one to run first.

Clauses (PreferJump1) and (PreferJump2) prioritize processes that can perform a jump. This helps collect jump instructions together so they are easier for post-fusion optimization to handle (§5.3). The instruction for the second process was computed by calling *tryStep* with the label arguments swapped, so in (PreferJump2) we need to swap the labels back with *swaplabels* (from Figure 13).

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

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

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

Figure 12 defines function *tryStep* which schedules a single instruction. This function takes the map of channel types, along with the current label and associated instruction of the first (left) process, and the current label of the other (right) process.

Clause (LocalJump) applies when the left process wants to jump. In this case, the result instruction simply performs the corresponding jump, leaving the right process where it is.

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

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

Clause (SharedPush) applies when the left process wants to push to a shared channel. Pushing to a shared channel requires the downstream process to be ready to accept the value at the same time. We encode this constraint by requiring the static input state of the downstream channel to be $none_F$. When this is satisfied, the result instruction stores the pushed value in the stream buffer variable (chan c) and sets the static input state to $pending_F$, which indicates that the new value is now available.

Still in Figure 12, clause (LocalPull) applies when the left process wants to pull from a local channel, which requires no coordination.

Clause (SharedPull) applies when the left process wants to pull from a shared channel that the other process either pulls from or pushes to. We know that there is already a value in the stream buffer variable, because the state for that channel is $pending_F$. The result instruction copies the value from the stream buffer variable into a variable specific to the left source process, and the corresponding $have_F$ channel state in the result label records that it has done so.

Clause (SharedPullInject) applies when the left process wants to pull from a shared channel that both processes pull from, and neither already has a value. The result instruction is a pull that loads the stream buffer variable.

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

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

Clauses (SharedDropOne) and (SharedDropBoth) apply when the left process wants to drop from a channel shared by both processes.

In (SharedDropOne) the channel states reveal that the other process is still using the value. In this case the result is a jump updating the channel state to note that the left process has dropped. In (SharedDropBoth) the channel states reveal that the other process no longer needs the value. In this case the result is a real drop, because we are sure that neither process requires the value any longer.

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

4.1 Fusibility

When we fuse a pair of processes we commit to a particular interleaving of instructions from each process. When we have at least three processes to fuse, the choice of which two to handle first can determine whether this fused result can then be fused with the third process. Consider the following example, where alt2 pulls two elements from its first input stream, then two from its second, before pushing all four to its output.

```
alternates : S Nat -> S Nat -> S Nat -> S (Nat, Nat)
alternates sInA sInB sInC
 = let   s1   = alt2 sInA sInB
         s2   = alt2 sInB sInC
         sOut = zip s1 s2
   in    sOut
```

If we fuse the two alt2 processes together first, then try to fuse this result process with the downstream zip process, the final fusion transform fails. This happens because the first fusion transform commits to a sequential instruction interleaving where two output values *must* be pushed to stream s1 first, before pushing values to s2. On the other hand, zip needs to pull a *single* value from each of its inputs alternately.

Dynamically, if we were to execute the first fused result process, and the downstream zip process concurrently, then the execution would deadlock. Statically, when we try to fuse the result process with the downstream zip process the deadlock is discovered and fusion fails. Deadlock happens when neither process can advance to the next instruction, and in the fusion algorithm this manifests as the failure of the *tryStepPair* function from Figure 11. The *tryStepPair* function determines which instruction from either process to execute next, and when execution is deadlocked there are none. Fusibility is an under-approximation for *deadlock freedom* of the network.

In practice, the likelihood of fusion succeeding depends on the particular dataflow network. For fusion of pipelines of standard combinators such as map, fold, filter, scan and so on, fusion always succeeds. The process implementations of each of these combinators only pull one element at a time from their source streams, before pushing the result to the output stream, so there is no possibility of deadlock. Deadlock can only happen when multiple streams fan-in to a process with multiple inputs, such as with merge.

When the dataflow network has a single output stream then we use the method of starting from the process closest to the output stream, walking towards the input streams, and fusing in successive processes as they occur. This allows the interleaving of the intermediate fused process to be dominated by the consumers, rather than producers, as consumers are more likely to have multiple input channels which need to be synchronized. In the worst case the fall back approach is to try all possible orderings of processes to fuse.

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

jump (l', u')	$\rightarrow$ Just (jump $\quad$ ((l', s_p), (l_q, s_q), u'))	(LocalJump)
case e (l_t', u_t') (l_f', u_f')	$\rightarrow$ Just (case e $\quad$ ((l_t', s_p), (l_q, s_q), u_t') ((l_f', s_p), (l_q, s_q), u_f'))	(LocalCase)

$\quad$ push c e (l', u')

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

$\quad$ pull c x (l', u')

$\mid cs[c] =$ in1	$\rightarrow$ Just (pull c x ((l', s_p), (l_q, s_q), u'))	(LocalPull)

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

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

$\quad$ drop c (l', u')

$\mid cs[c] =$ in1	$\rightarrow$ Just (drop c $\quad$ ((l', s_p), (l_q, s_q), u'))	(LocalDrop)
$\mid cs[c] =$ in1out1	$\rightarrow$ Just (jump $\quad$ (($l', s_p[c \mapsto$ none$_F$]), (l_q, s_q), u'))	(ConnectedDrop)
$\mid cs[c] =$ in2 $\wedge$ ($s_q[c] =$ have$_F \vee s_q[c] =$ pending$_F$)	$\rightarrow$ Just (jump $\quad$ (($l', s_p[c \mapsto$ none$_F$]), (l_q, s_q), u'))	(SharedDropOne)
$\mid cs[c] =$ in2 $\wedge$ $s_q[c] =$ none$_F$	$\rightarrow$ Just (drop c $\quad$ (($l', s_p[c \mapsto$ none$_F$]), (l_q, s_q), u'))	(SharedDropBoth)
_ $\quad \mid$ otherwise	$\rightarrow$ Nothing	(Blocked)

Figure 12: Fusion step for a single process of the pair.

channels $\quad$: $\quad$ *Process* $\rightarrow$ *Process* $\rightarrow$ (*Channel* $\mapsto$ *ChannelType2*)
channels p q $\quad$ = $\{c =$ in2 $\quad\mid c \in$ (ins $p \cap$ ins q)$\}$
$\qquad\qquad\qquad \cup$ $\{c =$ in1 $\quad\mid c \in$ (ins $p \cup$ ins q) $\wedge$ $c \notin$ (outs $p \cup$ outs q)$\}$
$\qquad\qquad\qquad \cup$ $\{c =$ in1out1 $\mid c \in$ (ins $p \cup$ ins q) $\wedge$ $c \in$ (outs $p \cup$ outs q)$\}$
$\qquad\qquad\qquad \cup$ $\{c =$ out1 $\quad\mid c \notin$ (ins $p \cup$ ins q) $\wedge$ $c \in$ (outs $p \cup$ outs q)$\}$

outlabels : *Instruction* $\rightarrow$ \{*Label*\}		*swaplabels* : *Instruction* $\rightarrow$ *Instruction*	
outlabels (pull c x (l, u))	= $\{l\}$	*swaplabels* (pull c x ((l_1, l_2), u))	= pull c x ((l_2, l_1), u)
outlabels (drop c (l, u))	= $\{l\}$	*swaplabels* (drop c ((l_1, l_2), u))	= drop c ((l_2, l_1), u)
outlabels (push c e (l, u))	= $\{l\}$	*swaplabels* (push c e ((l_1, l_2), u))	= push c e ((l_2, l_1), u)
outlabels (case e (l, u) (l', u'))	= $\{l, l'\}$	*swaplabels* (case e ((l_1, l_2), u) ((l_1', l_2'), u'))	= case e ((l_2, l_1), u) ((l_2', l_1'), u')
outlabels (jump (l, u))	= $\{l\}$	*swaplabels* (jump ((l_1, l_2), u))	= jump ((l_2, l_1), u)

Figure 13: Utility functions

5 IMPLEMENTATION

Stream fusion is ultimately performed for practical reasons: we want the fused result program to run faster than the original unfused program.

5.1 Finite streams

The processes we have seen so far deal with infinite streams, but in practice most streams are finite. Certain combinators such as fold and append only make sense on finite streams, and others like take produce inherently finite output. We have focussed on the infinite stream version as it is simpler to explain and prove, but supporting finite streams does not require substantial conceptual changes.

Unlike infinite streams, pulling from a finite stream can fail, meaning the stream is finished. We therefore modify the pull instruction to have two output labels: one to execute when a value is pulled, and the other to execute when the stream is finished. On the pushing

end, we also need some way of finishing streams, so we add a new instruction to close an output stream.

During evaluation we need some way of knowing whether a stream is closed, which can be added as an extra constructor in the *InputState* type. The same constructor is added to the static input state used by fusion. In this way, for any changes made to evaluation, the analogous static change must be made in the fusion transform.

It is also possible to encode finite streams as infinite streams with an explicit end-of-stream marker (EOF) and case statements. However, this requires the fusion transform to reason about case statements' predicates. By making the structure of finite streams explicit and constraining how processes use finite streams, it is not necessary to rely on heuristics for deciding equality of predicates.

This finite stream extension is described in more detail in the appendix of the extended version of this paper, which is available at http://cse.unsw.edu.au/~amosr/papers/merges.pdf.

5.2 Benchmarks

We have implemented this system using Template Haskell in a library called folderol[1]. To show practical examples, we use the finite stream extension mentioned in §5.1. We present three benchmarks: two file-based benchmarks, and one array algorithm.

For the file benchmarks, we compare against three Haskell streaming libraries: 'Conduit', 'Pipes', and 'Streaming'. These streaming libraries are pull-based, and do not naturally support multiple outputs: the split in the dataflow graph must be hand-fused, or somehow rewritten as a straight-line computation. These libraries also have a monadic interface, which allows the structure of the dataflow graph to depend on the values. This expressiveness has a price: if the dataflow graph can change dynamically, we cannot statically fuse it.

The first file benchmark simply appends two files while counting the lines. In Pipes and Conduit, counting the lines is implemented as a pipe which counts each line before passing it along. The first group in Figure 14 shows the runtimes for appending 2MB of data.

The second file benchmark takes a file and partitions it into two files: one with even-length lines, and one with odd-length lines. The output lines are also counted. Even with partial hand-fusion because of the multiple outputs, the Pipes and Conduit programs are slower than ours, as well as losing the abstraction benefits from using a high-level library. The 'Streaming' library allows streams to be shared in a fairly straightforward way and does not require hand-fusion, but is also the slowest in this benchmark. The second group in Figure 14 shows the runtimes for partitioning a 1MB file.

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

Here we also compare against a Data.Vector program, which uses shortcut fusion. The shortcut fusion system cannot fuse both operations into a single loop, and both operations must recompute the distances between the line and each point. As before, the Conduit and Pipes programs must be partially hand-fused. The third group in Figure 14 shows the runtimes for Quickhull over 40MB of data, or half a million points, while the final group uses more data (120MB) to compare directly against Data.Vector.

5.3 Optimisation and Drop Instructions

After we have fused two processes together, it may be possible to simplify the result before fusing in a third. Consider the result of fusing group and merge which we saw back in Figure 4. At labels F1 and F2 are two consecutive jump instructions. The update expressions attached to these instructions are also non-interfering, which means we can safely combine these instructions into a single jump. In general, we prefer to have jump instructions from separate processes scheduled into consecutive groups, rather than spread out through the result code. The (PreferJump) clauses of Figure 11 implement a heuristic that causes jump instructions to be scheduled before all others, so they tend to end up in these groups.

Other jump instructions like the one at F5 have no associated update expressions, and thus can be eliminated completely. Another simple optimization is to perform constant propagation, which in this case would allow us to eliminate the first case instruction.
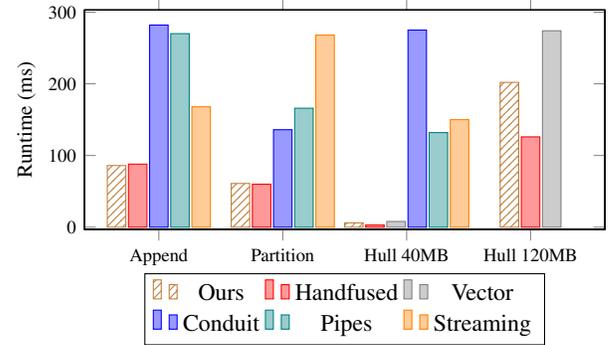
[1]https://github.com/amosr/folderol

Figure 14: Runtime for benchmarks; lower is faster.

Minimising the number of states in an intermediate process has the follow-on effect that the final fused result also has fewer states. Provided we do not change the order of instructions that require synchronization with other processes (pull, push or drop), the fusibility of the overall process network will not be affected.

When the two processes are able to accept the next variable from the stream at the same time, there is no need for the separate stream buffer variable. This is the case in Figure 4, and we can perform a copy-propagation optimisation, replacing all occurrences of v and x1 with the single variable b1. To increase the chance that we can perform copy-propagation, we need both processess to want to pull from the same stream at the same time. Moving the drop instruction for a particular stream as late as possible prevents a pull instruction from a second process being scheduled in too early.

6 PROOFS

Our fusion system is formalized in Coq, and we have proved soundness of *fusePair*: if the fused result process produces a particular sequence of values on its output channels then the two source processes may also produce that same sequence. The converse is not true, however: concurrent evaluation represents all possible interleavings, of which the fused process is only one.

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

EvalBs evaluates the fused program, and EvalOriginal ensures that the original program evaluates with that program's subset of the result heap. The Coq formalization has some small differences from the system in this paper. Instead of implementing non-deterministic evaluation we sequentially evaluate each source processes independently, and compare the output values to the ones produced by sequential evaluation of the fused result process. This is sufficient for our purposes because we are mainly interested in the value correctness of the fused program, rather than making a statement about the possible execution orders of the source processes when run concurrently.

7 RELATED WORK

This work aims to address the limitations of current combinator-based array fusion systems. As stated in the introduction, neither pull-based or push-based fusion is sufficient. Some combinators are inherently push-based, particularly those with multiple outputs such as unzip; while others are inherently pull-based, such as zip.

However, short cut fusion [10] relies on inlining which, like pull-based streams, only occurs when there is a single consumer. Push-based short cut fusion systems *do* exist, notably the original foldr/build formulation, but support neither zip nor unzip [19, 27].

Recent work on stream fusion [16] uses staged computation in a push-based system to ensure all combinators are inlined. When streams are used multiple times this causes excessive inlining, which duplicates work. This can change the semantics for effectful streams.

Data flow fusion [19] is neither pull-based nor push-based, and supports arbitrary splits and joins. It supports only a fixed set of standard combinators such as map, filter and fold, and converts each stream to a series with explicit rate types, similar to the clock types of Lucid Synchrone [2].

One way to address the difference between pull and push streams is to explicitly support both [3, 21]. Here, pull streams have the type Source and represent a source, always ready to be pulled, while push streams have the type Sink and represent a sink, always ready to accept data. The addition of push streams allows more programs to be fused than pull-only systems, but the computation must be manually split into sources and sinks.

The duality between pull and push arrays has also been explored in Obsidian [8, 28]. Here the distinction is made for the purpose of code generation for GPUs rather than fusion, as operations such as appending pull arrays require conditionals inside the loop, whereas using push arrays moves these conditionals outside the loop.

Meta-repa [1] supports both array types in a similar way, using Template Haskell for code generation. It supports fusion on both array types. When arrays are used multiple times, they must be explicitly reified with a 'force' operation to avoid duplication.

Streaming IO libraries have blossomed in the Haskell ecosystem, generally based on Iteratees [15]. Libraries such as conduit [25], enumerator [23], machines [17] and pipes [11] are all designed to write stream computations with bounded buffers, but do not guarantee fusion.

Like our own Machine Fusion, Filter Fusion [24] also statically interleaves the code of producer and consumer processes. Each process must have a single input and output channel, so common operators like zip, unzip, append, partition and so on are not supported. Given an adjacent producer and consumer pair, Filter Fusion alternately assigns control to the code of each. When the consumer needs input, control is passed to the producer, and when the producer produces its value control is passed back to the consumer. This simple scheduling algorithm works only for straight line pipelines of processes. Machine fusion provides a finer grained interleaving of code, which is nessesary to support branching dataflows that contain both splits and joins.

Synchronous languages such as LUSTRE [12], Lucy-n [22] and SIGNAL [18] all use some form of clock calculus and causality analysis to ensure that programs can be statically scheduled with bounded buffers [7]. These languages describe *passive* processes where values are fed in to streams from outside environments, such as data coming from sensors. In contrast, our processes are *active* and have control over when data is pulled from the source streams, which is nessesary for multiple input combinators such as merge and append. Synchronous dataflow languages reject operators with value dependent control flow such as merge, while general dataflow languages fall back on less performant dynamic scheduling [4].

Synchronous dataflow (SDF; not to be confused with synchronous languages above) is a dataflow graph model of computation where each node has constant, statically known input and output rates. StreamIt [29] uses synchronous dataflow for scheduling when possible, otherwise falling back to dynamic scheduling [26]. Boolean dataflow and integer dataflow [5, 6] extend SDF with boolean and integer valued control ports, but only support limited control flow.

REFERENCES

[1] Johan Ankner and Josef David Svenningsson. 2013. An EDSL approach to high performance Haskell programming. In *ACM SIGPLAN Notices*.
[2] Albert Benveniste, Paul Caspi, Stephen A Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* (2003).
[3] Jean-Philippe Bernardy and Josef Svenningsson. 2015. On the Duality of Streams How Can Linear Types Help to Solve the Lazy IO Problem?. In *IFL*.
[4] Adnan Bouakaz. 2013. *Real-time scheduling of dataflow graphs*. Ph.D. Dissertation. Université Rennes 1.
[5] Joseph T Buck. 1994. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *ACSSC*.
[6] Joseph Tobin Buck and Edward A Lee. 1993. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *ICASSP*.
[7] Paul Caspi and Marc Pouzet. 1996. Synchronous Kahn Networks. In *ICFP: International Conference on Functional Programming*.
[8] Koen Claessen, Mary Sheeran, and Joel Svensson. 2012. Expressive array constructs in an embedded GPU kernel programming language. In *DAMP*.
[9] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream fusion: From lists to streams to nothing at all. In *SIGPLAN*.
[10] Andrew Gill, John Launchbury, and Simon L Peyton Jones. 1993. A short cut to deforestation. In *FPCA*.
[11] Gabriel Gonzalez. 2012. http://hackage.haskell.org/package/pipes. (2012).
[12] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data flow programming language LUSTRE. *IEEE* (1991).
[13] Gilles Kahn, David MacQueen, and others. 1976. Coroutines and networks of parallel processes. (1976).
[14] Michael Kay. 2009. You pull, I'll push: on the polarity of pipelines. In *Balisage*.
[15] Oleg Kiselyov. 2012. Iteratees. In *FLOPS*.
[16] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. In *POPL*.
[17] Edward Kmett, Rúnar Bjarnason, and Josh Cough. 2012. http://hackage.haskell.org/package/machines. (2012).
[18] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. 2003. Polychrony for system design. *Journal of Circuits, Systems, and Computers* (2003).
[19] Ben Lippmeier, Manuel MT Chakravarty, Gabriele Keller, and Amos Robinson. 2013. Data flow fusion with series expressions in Haskell. In *SIGPLAN*.
[20] Ben Lippmeier, Manuel M. T. Chakravarty, Gabriele Keller, and Simon L. Peyton Jones. 2012. Guiding parallel array fusion with indexed types. In *Haskell*.
[21] Ben Lippmeier, Fil Mackay, and Amos Robinson. 2016. Polarized data parallel data flow. In *FHPC*.
[22] Louis Mandel, Florence Plateau, and Marc Pouzet. 2010. Lucy-n: a n-synchronous extension of Lustre. In *MPC*.
[23] John Millikin and Mikhail Vorozhtsov. 2011. http://hackage.haskell.org/package/enumerator. (2011).
[24] Todd A Proebsting and Scott A Watterson. 1996. Filter fusion. In *POPL*.
[25] Michael Snoyman. 2011. http://hackage.haskell.org/package/conduit. (2011).
[26] Robert Soule, Michael I. Gordon, Saman Amarasinghe, Robert Grimm, and Martin Hirzel. 2013. Dynamic Expressivity with Static Optimization for Streaming Languages. In *DEBS*.
[27] Josef Svenningsson. 2002. Shortcut fusion for accumulating parameters & zip-like functions. In *SIGPLAN*.
[28] Bo Joel Svensson and Josef Svenningsson. 2014. Defunctionalizing push arrays. In *FHPC*.
[29] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *Compiler Construction*.