Machine Fusion

Merging merges, more or less

AMOS ROBINSON, Ambiata and UNSW (Australia)
BEN LIPPMEIER, Digital Asset and UNSW (Australia)

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 and joins, and new operators can be added to the system without needing to modify the overall fusion transformation. Our system has been formalized in Coq, and we have proved soundness of the core transformation.

ACM Reference format:
Amos Robinson and Ben Lippmeier. 2016. Machine Fusion. 1, 1, Article 1 (January 2016), 28 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Suppose we have two input streams of numeric identifiers, and wish to perform some analysis on these identifiers. The identifiers from both streams arrive 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 identifiers from 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. If all values from `sIn1` are smaller than the ones from `sIn2`, then `merge` will pull all values from `sIn1` before pulling any from `sIn2`, and vice versa. Secondly, although `sIn1` occurs twice in the program, at runtime we only want to handle the elements of each stream 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 the `group` and `merge` operators are ready to receive it. Finally, as the stream length is unbounded, we cannot buffer an arbitrary number of elements read from either stream, or risk running out of local storage space.

For an implementation, we might try coding each of the operators as a separate concurrent process, and send each stream element using an intra-process communication mechanism. Developing such an implementation could be easy or hard, depending on what language features are available for concurrency. However, worrying about the *performance tuning* of such a system, such as whether we need back-pressure to prevent buffers from being overrun, or how to chunk the stream data to reduce the amount of communication overhead, is invariably a headache.

2016. XXXX-XXXX/2016/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

We might instead define some sort of uniform sequential interface for data sources, with a single 'pull' function that provides the next value in each stream. Each operator could be given this interface, so that the next value from each result stream is computed on demand. This approach is commonly taken with implementations of physical operators in database systems (Graefe 1994). However, the 'pull only' model does not support operators with multiple outputs, such as our derived `uniquesUnion` operator, at least not without unbounded buffering. Suppose a consumer pulls many elements from the `sUnique` output stream. In order to perform the `group` operation, the implementation needs to pull the corresponding source elements from the `sIn1` input stream *as well* as buffering an arbitrary number of them. It needs to buffer these elements because they are also needed to perform the `merge` operation. When a consumer finally pulls from `sUnion` we will be able to drain the buffer of `sIn1` elements, but not before.

Instead, for a single threaded program, we want to perform *stream fusion*, which takes the implied dataflow network and produces a simple sequential loop that gets the job done without requiring extra process-control abstractions and without requiring unbounded buffering. Sadly, existing stream fusion transformations cannot handle our example. As observed by Kay (2009), both pull-based and push-based fusion have fundamental limitations. Pull-based systems such as short-cut stream fusion (Coutts et al. 2007) cannot handle cases where a particular stream or intermediate result is used by multiple consumers. We refer to this situation as a *split* — in the dataflow network of our example the flow from input stream `sIn1` is split into both the `group` and `merge` consumers.

Push-based systems such as foldr/build fusion (Gill et al. 1993) cannot fuse our example either, because they do not support operators with multiple inputs. We refer to such a situation as a *join* — in our example the `merge` operator expresses a join in the data-flow network. Some systems support both pull and push: data flow inspired array fusion using series expressions (Lippmeier et al. 2013) allows both splits and joins but only for a limited, predefined set of operators. More recent work on polarized data flow fusion (Lippmeier et al. 2016) *is* able to fuse our example, but requires the program to be rewritten to use explicitly polarized stream types.

Synchronous dataflow languages such as Lucy-n (Mandel et al. 2010) reject value-dependent operators with value dependent control flow such as `merge`, while general dataflow languages fall back on less performant dynamic scheduling for these cases (Bouakaz 2013). The polyhedral array fusion model (Feautrier and Lengauer 2011) is used for loop transformations in imperative programs, but operates at a much lower level. The polyhedral model is based around affine loops, which makes it difficult to support filter-like operators such as `group` and `merge`.

In this paper we present Machine Fusion, a new approach. Each operator is expressed as a restricted, sequential imperative program with commands that include both `pull` for reading from an input stream and `push` for writing to an output stream. We view each operator as a process in a concurrent process network, and the control flow of each process as a simple state machine. 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 the fusion transform succeeds we know it has worked. There is no need to inspect intermediate representations of the compiler to debug poor performance, which is a common problem in systems based on general purpose program transformations (Lippmeier et al. 2012).

In summary, we make the following contributions:

- a process calculus for encoding infinite streaming programs (§2);
- an algorithm for fusing these processes, the first to support splits and joins (§4);
- numerical results that demonstrate that the algorithm is well behaved when the number of fused processes is large. The size of the fused result program is not excessive. (§5);
- a formalization and proof of soundness for the core 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 §8.2.

2 PROCESSES, MACHINES, COMBINATORS AND OPERATORS

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 particular 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 Fig. 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 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 of the record 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. Note that in this form the output stream is defined via a parameter, rather than being the result of the combinator, as in the source 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 `(push sOut1 v A3 [ l = v, f = F ])` 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 the instruction at `A0`. This `drop` instruction is used to coordinate concurrent processes when performing fusion. The next element of a stream may only be pulled after all consumers of that stream have pulled and then and dropped the current element.

2.2 Merging

The definition of the `merge` combinator, which merges two input streams, is given in Fig. 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` are used to 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.

```
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 [] } }
```

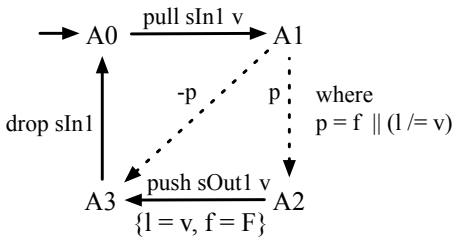

Fig. 1. The group combinator

As the `merge` process merges infinite streams, if we execute it with a finite input prefix, it will arrive at an intermediate state that may not yet have pushed all available output. For example, if we execute the process with the input streams `[1,4]` and `[2,3,100]` then the values `[1,2,3,4]` will be pushed to the output. After pushing the last value 4, the process will block at instruction `E2`, waiting for the next value to become available from `sIn2`. We discuss how to handle finite streams later in §8.2.

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. For ease of reference, when describing 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 in 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 Fig. 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`.

```
merge
  = λ (sIn1: Stream Nat) (sIn2: Stream Nat) (sOut2: Stream Nat).
    ν (x1: Nat) (x2: Nat) (B0..E2: Label).
    process
     { ins:    { sM1, sM2 }
     , outs:   { sM3 }
     , 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 [] } }
```

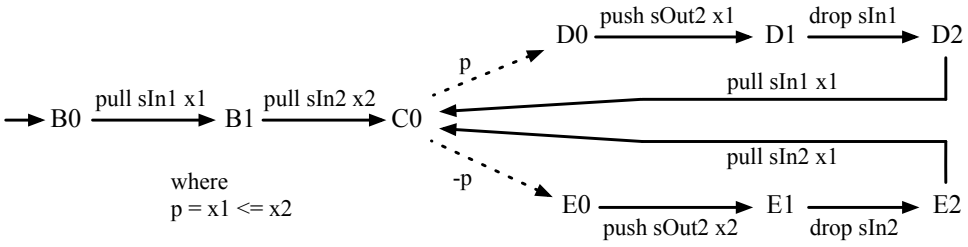

Fig. 2. The merge combinator

We also associate each of the joint result states with information describing whether or not each source state machine has already pulled a value from each of its input streams. For the `F0` case shown in Fig. 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 transition 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. In Fig. 3, 'sIn1' is set to 'have' for the first machine in `F2` after we have set 'v = b1', while 'sIn1' is set to 'have' for the second machine in `F3` after we have set 'x1 = b1'.

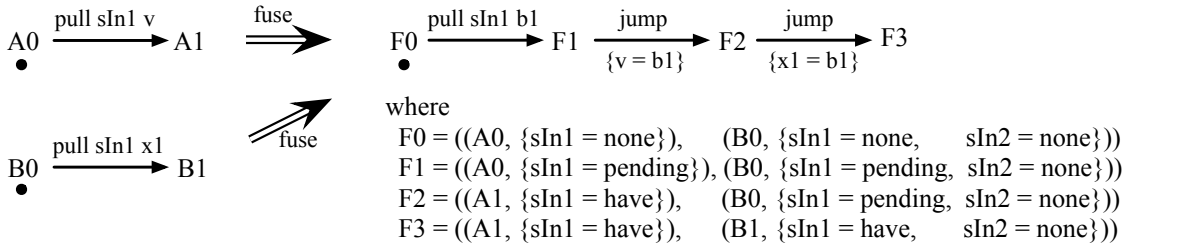

Fig. 3. Fusing pull instructions

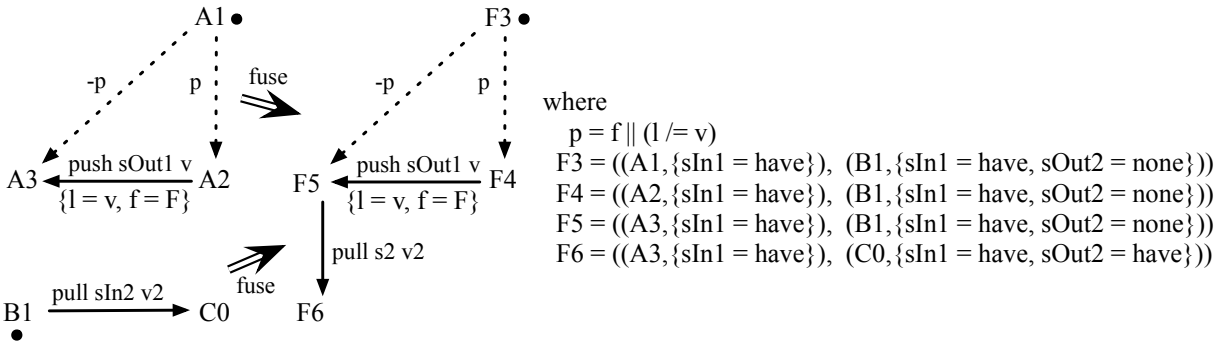

Fig. 4. Fusing case instructions

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 left of Fig. 4 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 `A2` or `A3`, corresponding to `F4` and `F5` respectively. From state `F4`, the push at `A2` is executed and moves to `A3`, corresponding to `F5`.

Finally, at `F5` the `merge` machine pulls from `sIn2`, moving from `B1` to `C0`. Because the stream `sIn2` is only pulled from by the `merge` machine, no coordination is required between the `merge` and `group` machines for this pull.

Note that we could construct the fused result machine in several ways. One option is to perform the case branch first and then pull from `sIn2`, another is to pull from `sIn2` first and then perform the branch. By construction, the predicate used in the branch refers only to variables local to the `group` machine, and the pull instruction from `B1` stores its result in a variable local to the `merge` machine. As the set of variables does not overlap, either ordering is correct. For this example we choose to perform the branch first, though will discuss the ramifications of this choice further in §5.1.

2.4 Fused Result

Fig. 5 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` which is the `group` processes output stream, and `sOut2` which is the `merge` processes output stream.

```
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}))
} }
```

Fig. 5. Fusion of group and merge, along with shared instructions

Overall, our fusion algorithm has taken two separate processes that we once imagined to be running concurrently, and has produced a single sequential result process that implements both. We have chosen a *specific sequential order* in which to interleave the instructions that implement both source processes. As with the Flow Fusion system of Lippmeier et al. (2013) we have performed the job of a concurrent scheduler at compile time. However, in contrast to Flow Fusion and similar systems, 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 processes, including ones with nested loops that work on segmented streams, which we discuss further in §8.

To complete the implementation of our example from §1 we would now proceed to fuse the process from the final line (also a `group`) into this new result process. The order in which pairs of processes are fused together does matter, as does the order in which the instructions are interleaved — we discuss both points further in §5.

Finally, although the result process has a single shared heap, the bindings are guaranteed not to interfere. When we instantiated combinators to create the original source processes we introduced fresh names at that point. The stream buffer variables we additionally introduced for coordination were freshly created during fusion.

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

$$
\begin{array}{lll}
\textit{Variable}, x & \rightarrow & \text{(value variable)} \\
\textit{Channel}, c & \rightarrow & \text{(channel name)} \\
\textit{Label}, l & \rightarrow & \text{(label name)} \\
\textit{ChannelStates} & = & (\textit{Channel} \mapsto \textit{InputState})
\end{array}
$$

$$
\begin{array}{lll}
\textit{Action}, a & ::= & \cdot \mid \texttt{push}\ \textit{Channel}\ \textit{Value}
\end{array}
$$

$$
\begin{array}{lll}
\textit{Process}, p & ::= & \texttt{process} \\
 & & \quad \texttt{ins:}\ \ (\textit{Channel} \mapsto \textit{InputState}) \\
 & & \quad \texttt{outs:}\ \ \{\textit{Channel}\} \\
 & & \quad \texttt{heap:}\ \ \textit{Heap} \\
 & & \quad \texttt{label:}\ \ \textit{Label} \\
 & & \quad \texttt{instrs:}\ (\textit{Label} \mapsto \textit{Instruction})
\end{array}
$$

$$
\begin{array}{lll}
\textit{Instruction} & ::= & \texttt{pull}\ \textit{Channel}\ \textit{Variable}\ \textit{Next} \\
 & & \mid\ \texttt{push}\ \textit{Channel}\ \textit{Exp}\ \ \ \ \textit{Next} \\
 & & \mid\ \texttt{drop}\ \textit{Channel}\ \ \ \ \ \ \ \ \textit{Next} \\
 & & \mid\ \texttt{case}\ \textit{Exp}\ \ \textit{Next}\ \ \textit{Next} \\
 & & \mid\ \texttt{jump}\ \ \ \ \ \ \ \ \ \ \ \ \textit{Next}
\end{array}
$$

$$
\begin{array}{lll}
\textit{InputState} & ::= & \texttt{none} \mid \texttt{pending}\ \textit{Value} \mid \texttt{have} \\
\textit{Next} & = & \textit{Label} \times \textit{Updates}
\end{array}
$$

Fig. 6. Process definitions

3 PROCESS DEFINITIONS

The grammar for process definitions is given in Fig. 6. Variables, Channels and Labels are specified by unique names. We refer to the *endpoint* of a stream as a channel. A particular stream may flow into the input channels of several different processes, but can only 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-then.

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'.

In the grammar of Fig. 6 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 used a version of *InputState* with only this staticaly 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.

All instructions include a *Next* field which is a pair of 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 `drops` instructions are used to control how processes should be synchronized, but do not affect the execution of a single process. We will discuss `drops` further in §5.3.

3.1 Execution

The dynamic execution of process networks consists of three aspects:

(1) *Injection* of a single value from a stream into a process, or a set of processes. Each individual process only needs to accept the value when it is ready for it, and injection of a value into a set of processes succeeds only when they *all* accept it.
(2) *Advancing* a single process from one state to another. Advancing a set of processes suceeds when *any* of the processes in the set 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 then 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 process network is non-deterministic. At any moment several processes may be able to take a step, while others are blocked. As with Kahn processes (Kahn et al. 1976), pulling from a channel is blocking. This 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, which is the stream buffer variable, and pushing can only succeed 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 exploit this fact by choosing one particular instruction ordering that enables the process network to advance without requiring unbounded buffering.

Each output 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" (or perhaps "functional data flow") as there are no side-channels between the processes. This is in contrast to systems such as StreamIt (Thies et al. 2002), where the processes are also able to send asynchronous messages to each other, in addition to via the formal input and output streams.

3.1.1 Injection. Fig. 7 gives the rules for injecting values into processes. 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, but operates on sets of processes rather than a single one.

Rule (InjectValue) injects of 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 (set to `none`) for the 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 set of processes. We use the single process judgment form to inject the value into all processes in the set, which must succeed for all of them. Once a value has been injected into all consuming processes that require it, the producing process no longer needs to retain it.

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

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

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

Fig. 7. Injection of values into input channels

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

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

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

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

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

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

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

Fig. 8. Advancing processes

3.1.2 Advancing. Fig. 8 gives the rules for advancing a single process. The first set of rules handle specific instructions. The statement $(i;\ is;\ bs \xRightarrow{a} l;\ is';\ us')$ 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.

$$\boxed{\{Process\} \overset{Action}{\Longrightarrow} \{Process\}}$$

$$\frac{p_i \overset{\cdot}{\Rightarrow} p_i'}{\{p_0 \ldots p_i \ldots p_n\} \overset{\cdot}{\Rightarrow} \{p_0 \ldots p_i' \ldots p_n\}} \text{ (ProcessesInternal)}$$

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

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

$$\frac{ps \overset{\cdot}{\Rightarrow} ps'}{cvs;\ ps \Rightarrow cvs;\ ps'} \text{ (FeedInternal)}$$

$$\frac{ps \overset{\text{push } c\ v}{\Longrightarrow} ps'}{cvs;\ ps \Rightarrow cvs[c \mapsto (cvs[c]\ \texttt{++}\ v)];\ ps'} \text{ (FeedPush)}$$

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

Fig. 9. Feeding Process Networks

Rule (Pull) takes the `pending` value v from the channel state and produces a heap update that will copy this value into the variable x named 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 and 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 lookup the current instruction pointed to by the processes `label` and pass it, along with the current channel states and heap to the previous single instruction judgment. The update expressions `us` that the single instruction judgment yields are first reduced to values before updating the heap. We use $(us \triangleleft bs)$ to replace bindings in us with new ones from bs. The update expressions themselves are all pure, so the evaluation can safely be done in parallel (or in arbitrary order).

3.1.3 Feeding. Fig. 9 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 process network, while the second exchanges input and output values with the environment the process network is running in.

The statement $ps \stackrel{a}{\Rightarrow} ps'$ reads "the processes group ps advances to the new process group ps' yielding output action a. A process "group" is just a set of processes.

Rule (ProcessInternal) allows an arbitrary process in the group to advance to a new state at any time, provided it does not yield an output action. This allows processes to perform internal computation, without needing to synchronize with the rest of the group.

Rule (ProcessPush) allows an arbitrary process in the group to advance to a new state while yielding an output action (push c v). For this to happen it must be possible to inject the output value v into all processes that have channel c as one of their inputs. As all consuming processes must accept the output value at the time it is created, there is no need to buffer it further in the producing process. When any process in the group produces an output action we take that as the action of the whole group.

The statement $cvs;\ ps \Rightarrow cvs';\ ps'$ reads "with channel values cvs, process group ps takes a step and produces new channel values cvs' and group ps'". The channel values cvs map channel names to a list of values. For input channels of the overall group, we initialize the map to contain a list of input values for each channel. For output channels of the overall group, 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 here is sufficient to describe the abstract behavior of our system.

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

Rule (FeedPush) collects an output action (push c v) produced by a process group 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 used during formal proofs of correctness, but would not need to be checked dynamically in a concrete implementation. The topology of the dataflow network does not change at runtime, so it only needs to be checked once, before execution.

3.2 Non-deterministic Execution Order

The execution rules of Fig. 9 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 group. More importantly, (ProcessPush) allows any process to perform a push action at any time, provided all other processes in the group are ready to accept the pushed value. Rule (FeedExternal) also allows new values to be injected from the environment, provided all processes that use the associated channel are ready to accept the value.

In our system, allowing the execution order of processes to be non-deterministic is critical, as it provides freedom to search for a valid ordering that does not require excessive buffering. Consider the following example, where the `alt2` operator pulls two elements from its first input stream, then two from the second, before pushing all four to its output stream.

```
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
```

Note that the middle stream `sInB` is shared, and the result streams from both `alt2` operators are zipped into tuples. Given the inputs `sInA = [a1,a2]`, `sInB = [b1,b2]` and `sInC = [c1,c2]` the output of `zip` will be `[(a1,b1),(a2,b2),(b1,c1),(b2,c2)]`, assuming `a1,a2,b1,b2` and so on are values of type `Nat`.

Now, note that the first `alt2` process pushes values to its output stream `s1` two at a time, and the second `alt2` process also pushes values to its own output stream `s2` two at a time. However, the downstream `zip` process needs to pull one value from `s1` then one from `s2`, then another from `s1`, then another from `s2`, alternating between the `s1` and `s2` streams. This will work, provided we can arrange for the two *separate* `alt2` processes to push to their separate output streams alternatively. They can still push two values at a time to their own outputs, but the downstream `zip` process needs receive one from each process alternately. Here is a table of intermediate values to help make the explanation clearer:

```
sInA = [a1, a2, a3, a4, a5 ...]
sInB = [b1, b2, b3, b4, b5 ...]
sInC = [c1, c2, c3, c4, c5 ...]

s1   = alt2 sInA sInB
     = [a1, a2, b1, b2, a3, a4, b3, b4 ...]

s2   = alt2 sInB sInC
     = [b1, b2, c1, c2, b3, b4, c3, c4 ...]

sOut = zip s1 s2
     = [(a1,b1), (a2,b2), (b1,c1), (b2,c2) ...]
```

Considering the last line in the above table, note that `zip` needs to output a tuple of `a1` and `b1` together, then `a2` and `b2` together, and so on. The implementation of the `zip` process will attempt to pull the first value `a1` from stream `s1`, blocking until it gets it, then pull the next value `b1` from stream `s2`, blocking until it gets it. While `zip` is blocked waiting for `b1`, the first `alt2` process cannot yet push `a2`. The execution order of the overall process group is constrained by communication patterns of processes in that group.

As we cannot encode all possible orderings into the definition of the processes themselves, we have defined the execution rules to admit many possible orderings. In 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 transform 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 process group was executed dynamically. In doing so, we also eliminate the need to pass messages between processes — once they are fused we can just copy values between heap locations.

4 FUSION

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

Fig. 10 defines some auxiliary grammar used during fusion. We extend the exiting *Label* grammar with an additional alternative, $Label_S \times Label_S$ which we use for the labels in a fused result process. Each $Label_S$ consists of a *Label* from one of the source processes, paired with a map from *Channel* to the statically known part of that channel's current *InputState*. The definition of *Label* is now recursive. While fusing a whole process group, 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 stream buffer variable associated with channel `c`. We only need one stream buffer variable for each channel, and naming them like this saves us from inventing fresh variable names in the definition of the fusion rules (we used a fresh name `b1` back in §2.3.1 to avoid introducing a new mechanism at that point in the discussion)

$$
\begin{array}{lll}
\mathit{Label} & ::= & \ldots \mid \mathit{Label}_S \times \mathit{Label}_S \mid \ldots \\
\mathit{Label}_S & = & \mathit{Label} \times (\mathit{Channel} \mapsto \mathit{InputState}_S) \\
\mathit{InputState}_S & ::= & \mathsf{none}_S \mid \mathsf{pending}_S \mid \mathsf{have}_S \\
\mathit{Variable} & ::= & \ldots \mid \mathsf{chan}\ \mathit{Channel} \mid \ldots \\
 & & \\
\mathit{ChannelType}_2 & ::= & \mathtt{in2} \mid \mathtt{in1} \mid \mathtt{in1out1} \mid \mathtt{out1}
\end{array}
$$

Fig. 10. Fusion type definitions.

$$
\begin{array}{l}
\mathit{fusePair} \quad : \quad \mathit{Process} \to \mathit{Process} \to \mathit{Maybe}\ \mathit{Process} \\
\mathit{fusePair}\ p\ q \ = \\
\quad \mathtt{process} \\
\qquad \mathtt{ins:} \quad \{c \mid c = t \in cs,\ t \in \{\mathtt{in1}, \mathtt{in2}\}\} \\
\qquad \mathtt{outs:} \quad \{c \mid c = t \in cs,\ t \in \{\mathtt{in1out1}, \mathtt{out1}\}\} \\
\qquad \mathtt{heap:} \quad p[\mathtt{heap}] \cup q[\mathtt{heap}] \\
\qquad \mathtt{label:} \quad l_0 \\
\qquad \mathtt{instrs:} \quad \mathit{go}\ \{\}\ l_0 \\
\quad \mathtt{where} \\
\qquad cs \ = \ \mathit{channels}\ p\ q \\
\qquad l_0 \ = \ \big((p[\mathtt{label}],\ \{c = \mathit{none}_S \mid c \in p[\mathtt{ins}]\}),\ (q[\mathtt{label}],\ \{c = \mathit{none}_S \mid c \in q[\mathtt{ins}]\})\big) \\
 \\
\qquad \mathit{go}\ \mathit{bs}\ (l_p, l_q) \\
\qquad\quad \mid (l_p, l_q) \in \mathit{bs} \\
\qquad\quad = \mathit{bs} \\
\qquad\quad \mid b \in \mathit{tryStepPair}\ cs\ l_p\ p[\mathtt{instrs}][l_p]\ l_q\ q[\mathtt{instrs}][l_q] \\
\qquad\quad = \mathit{fold}\ \mathit{go}\ (\mathit{bs} \cup \{(l_p, l_q) = b\})\ (\mathit{outlabels}\ b)
\end{array}
$$

Fig. 11. Fusion of pairs of processes

Still in Fig. 10, $\mathit{ChannelType}_2$ classifies how channels are used, and possibly shared, between two processes. Type `in2` indicates that the two processes `pull` from the same channel, and these actions must then 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 from the same. 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.

Fig. 11 defines function *fusePair* that fuses a pair of processes together, 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 Fig. 14) define new joint labels and reachable states. As we discover reachable states we add them to a map *bs* of joint label to corresponding the instruction, and repeat the process to a fixpoint where no more states can be discovered.

$$
\begin{aligned}
&\mathit{tryStepPair} : (\mathit{Channel} \mapsto \mathit{ChannelType}_2) \to \mathit{Label}_1 \to \mathit{Instruction} \to \mathit{Label}_1 \to \mathit{Instruction} \to \mathit{Maybe\ Instruction} \\
&\mathit{tryStepPair}\ cs\ l_p\ i_p\ l_q\ i_q \\
&\quad \text{(PreferJump1)} \\
&\quad \mid i'_p \in \mathit{tryStep}\ cs\ l_p\ i_p\ l_q \wedge \texttt{jump}\ (l,u) \in i'_p \to i'_p \\
&\quad \text{(PreferJump2)} \\
&\quad \mid i'_q \in \mathit{tryStep}\ cs\ l_q\ i_q\ l_p \wedge \texttt{jump}\ (l,u) \in i'_q \to \mathit{swaplabels}\ i'_q \\
&\quad \text{(DeferPull1)} \\
&\quad \mid i'_p \in \mathit{tryStep}\ cs\ l_p\ i_p\ l_q \wedge i'_q \in \mathit{tryStep}\ cs\ l_q\ i_q\ l_p \wedge \texttt{pull}\ c\ x\ (l,u) \notin i'_p \to i'_p \\
&\quad \text{(DeferPull2)} \\
&\quad \mid i'_p \in \mathit{tryStep}\ cs\ l_p\ i_p\ l_q \wedge i'_q \in \mathit{tryStep}\ cs\ l_q\ i_q\ l_p \wedge \texttt{pull}\ c\ x\ (l,u) \notin i'_q \to \mathit{swaplabels}\ i'_q \\
&\quad \text{(Run1)} \\
&\quad \mid i'_p \in \mathit{tryStep}\ cs\ l_p\ i_p\ l_q \to i'_p \\
&\quad \text{(Run2)} \\
&\quad \mid i'_q \in \mathit{tryStep}\ cs\ l_q\ i_q\ l_p \to \mathit{swaplabels}\ i'_q
\end{aligned}
$$

Fig. 12. Fusion step coordination for a pair of processes.

Fig. 12 defines function *tryStepPair* which decides which of two processes to advance.

Clauses (PreferJump1) and (PreferJump2) are heuristics which prioritize processes that can perform a `jump`. This heuristic helps to collect `jump` instructions together in the result process, which are then easier for post-fusion optimization (§5.3) to handle. As clause (PreferJump2) calls *tryStep* with the instructions in swapped order, if *tryStep* succeeds we use *swaplabels* (from Fig. 14) to re-swap the labels in the result.

Clauses (DeferPull1) and (DeferPull2) are similar heuristics, which defer `pull` instructions if possible. We do this because `pull` instructions may block, while other instructions are more likely to produce immediate results.

Clauses (Run1) and (Run2) apply when one process can advance using some other instruction. We try the first process first, and if that can advance then so be it. This priority means that fusion is left-biased, preferring advancement of the left process over the second.

Fig. 13 defines function *tryStep* which schedules a single instruction. This function takes the map of channel types, along with the current label and associated instruction of the first (left) process, and the current label of the other (right) process.

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 that there are two destination 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 $\texttt{none}_S$. 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 "$\texttt{pending}_S$", which indicates that the new value is now available.

```
tryStep : (Channel ↦ ChannelType2) → Label1 → Instruction → Label1 → Maybe Instruction
tryStep cs (lp,sp) ip (lq,sq) = match ip with

  jump (l',u')
    (LocalJump)
    → jump ((l', sp), (lq, sq), u')

  case e (lt',ut') (lf',uf')
    (LocalCase)
    → case e ((lt', sp), (lq, sq), ut') ((lf', sp), (lq, sq), uf')

  push c e (l',u')
    (LocalPush)
    | cs[c] = out1
    → push c e ((l', sp), (lq, sq), u')
    (SharedPush)
    | cs[c] = in1out1 ∧ sq[c] = noneS
    → push c e ((l', sp), (lq, sq[c ↦ pendingS]), u'[chan c ↦ e])

  pull c x (l',u')
    (LocalPull)
    | cs[c] = in1
    → pull c x ((l', sp), (lq, sq), u')
    (SharedPull)
    | (cs[c] = in2 ∨ cs[c] = in1out1) ∧ sp[c] = pendingS
    → jump ((l', sp[c ↦ haveS]), (lq, sq), u'[x ↦ chan c])
    (SharedPullInject)
    | cs[c] = in2 ∧ sp[c] = noneS ∧ sq[c] = noneS
    → pull c (chan c) ((lp, sp[c ↦ pendingS]), (lq, sq[c ↦ pendingS]), [])

  drop c (l',u')
    (LocalDrop)
    | cs[c] = in1
    → drop c ((l', sp), (lq, sq), u')
    (ConnectedDrop)
    | cs[c] = in1out1
    → jump ((l', sp[c ↦ noneS]), (lq, sq), u')
    (SharedDropOne)
    | cs[c] = in2 ∧ (sq[c] = haveS ∨ sq[c] = pendingS)
    → jump ((l', sp[c ↦ noneS]), (lq, sq), u')
    (SharedDropBoth)
    | cs[c] = in2 ∧ sq[c] = noneS
    → drop c ((l', sp[c ↦ noneS]), (lq, sq), u')
```

Fig. 13. Fusion step for a single process of the pair.

$$
\begin{array}{lcl}
\mathit{channels} & : & \mathit{Process} \to \mathit{Process} \to (\mathit{Channel} \mapsto \mathit{ChannelType}_2) \\
\mathit{channels}\ p\ q & = & \{c = \mathtt{in2} \mid c \in (\mathtt{ins}\ p \cap \mathtt{ins}\ q)\} \\
& \cup & \{c = \mathtt{in1} \mid c \in (\mathtt{ins}\ p \cup \mathtt{ins}\ q) \wedge c \notin (\mathtt{outs}\ p \cup \mathtt{outs}\ q)\} \\
& \cup & \{c = \mathtt{in1out1} \mid c \in (\mathtt{ins}\ p \cup \mathtt{ins}\ q) \wedge c \in (\mathtt{outs}\ p \cup \mathtt{outs}\ q)\} \\
& \cup & \{c = \mathtt{out1} \mid c \notin (\mathtt{ins}\ p \cup \mathtt{ins}\ q) \wedge c \in (\mathtt{outs}\ p \cup \mathtt{outs}\ q)\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{outlabels} & : & \mathit{Instruction} \to \{\mathit{Label}\} \\
\mathit{outlabels}\ (\mathtt{pull}\ c\ x\ (l,u)) & = & \{l\} \\
\mathit{outlabels}\ (\mathtt{drop}\ c\ (l,u)) & = & \{l\} \\
\mathit{outlabels}\ (\mathtt{push}\ c\ e\ (l,u)) & = & \{l\} \\
\mathit{outlabels}\ (\mathtt{case}\ e\ (l,u)\ (l',u')) & = & \{l,l'\} \\
\mathit{outlabels}\ (\mathtt{jump}\ (l,u)) & = & \{l\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{swaplabels} & : & \mathit{Instruction} \to \mathit{Instruction} \\
\mathit{swaplabels}\ (\mathtt{pull}\ c\ x\ ((l_1,l_2),u)) & = & \mathtt{pull}\ c\ x\ ((l_2,l_1),u) \\
\mathit{swaplabels}\ (\mathtt{drop}\ c\ ((l_1,l_2),u)) & = & \mathtt{drop}\ c\ ((l_2,l_1),u) \\
\mathit{swaplabels}\ (\mathtt{push}\ c\ e\ ((l_1,l_2),u)) & = & \mathtt{push}\ c\ e\ ((l_2,l_1),u) \\
\mathit{swaplabels}\ (\mathtt{case}\ e\ ((l_1,l_2),u)\ ((l'_1,l'_2),u')) & = & \mathtt{case}\ e\ ((l_2,l_1),u)\ ((l'_2,l'_1),u') \\
\mathit{swaplabels}\ (\mathtt{jump}\ ((l_1,l_2),u)) & = & \mathtt{jump}\ ((l_2,l_1),u)
\end{array}
$$

Fig. 14. Utility functions

Still in Fig. 13, 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 $\mathtt{pending}_S$. The result instruction copies the value from the stream buffer variable into a variable specific to the left source process, and the corresponding $\mathtt{have}_S$ 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 from 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 a value it gained from a channel which is shared by the other process. In (SharedDropOne) the channel states reveal that the other process is still using the value. In this case the result instruction is a `jump` that only updates the static information that the left source process no longer requires this value. In (SharedDropBoth) then channel states reveal that the other process no longer needs the value either. In this case the result instruction is a real `drop`, because we are sure that neither process requires the value any longer.

Fig. 14 contains definitions of some utility functions which we have already mentioned. Function *channels* computes the $\mathit{ChannelType}_2$ map for a pair of processes. Function *outlabels* gets the set of output labels for an instruction, which is used when computing the fixpoint of reachable states. Function *swaplabels* flips the order of the compound labels in an instruction.

5 FUSION IN PRACTICE

Stream fusion is ultimately performed for practical reasons. We want the fused result program to run faster than unfused source program. Fused result programs like the one in Fig. 5 can be lowered directly to a target language, such as Haskell (maybe using Template Haskell), C or LLVM abstract assembly. The details of such lowering surely affect the performance of the final program, but as they are largely unrelated to the fusion algorithm itself, we focus on the form of the fused program expressed directly in the process language.

5.1 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 our `alternatives` example from §3.2, here it is again:

```
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
```

In our current system, if we fuse the two `alt2` processes together first, then try to fuse this result process with the downstream `zip` process, then this 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 then pushing values to `s2`. As we discussed in §3.2, the downstream `zip` process needs to pull single values from `s1` and `s2` alternatively.

Dynamically, if we were to try to execute the 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 then the deadlock is discovered and fusion fails. Deadlock happens when neither process can advance to the next instruction, and in the fusion algorithm this will manifest as the failure of the *tryStepPair* function from Fig.12. The *tryStepPair* function determines which of the instructions from either process can be executed next, and when execution is deadlocked there are none.

On the upside, fusion failure is easy to detect. It is also easy to provide a report to the client programmer that describes why two particular processes could not be fused. The report is phrased in terms of the process definitions visible to the client programmer, instead of partially fused intermediate code. The joint labels used in the fusion algorithm represent which states each of the original processes would be in during a concurrent execution, and we provide the corresponding instructions as well as the abstract states of all the input channels. This reporting ability is *significantly better* than that of prior fusion systems such as Repa (Lippmeier et al. 2012), as well as the co-recursive stream fusion of (Coutts et al. 2007), and many other systems based on general purpose program transformations. In such systems it is usually not clear whether the fusion transformation even succeeded, and debugging why it might not have succeeded involves spelunking[1] through many pages (sometimes hundreds of pages) of compiler intermediate representations.

In practice, the likelihood of fusion suceeding depends on the particular dataflow network being used, as well as the form of the processes in that 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 possiblity 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

[1] def. spelunking: Exploration of caves, especially as a hobby. Usually not a science.

the process closest to the output stream, walking to 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 fallback approach is to try all possible orderings of processes to fuse, assuming the client programmer is willing to wait for the search to complete.

5.2 Result Size

As with any fusion system, we must be careful that the size of the result code does not become too large when more and more processes are fused together. The left of Fig. 15 shows the maximum number of output states in the result when a particular number of processes are fused together in a pipelined-manner. To produce this graph we programmatically generated dataflow networks for *all possible* pipelined combinations of the `map`, `filter`, `scan`, `group` and `merge` combinators, and tried all possible fusion orders consiting of adjacent pairs of processes. The `merge` combinator itself has two inputs, so only works at the very start of the pipeline — we present result for pipelines with and without a `merge` at the start. The right of Fig. 15 shows the number of states in the result when the various combinations of combinators are fused in parallel, for example, we might have a `map` and a `filter` processing the same input stream. In both cases the number of states in the result process grows linearly with the number of processes. In all combinations, with up to 7 processes there are less than 100 states in the result process.

The size of the result process is roughly what one would get when inlining the definitions of each of the original source processes. This is common with other systems based on inlining and/or template meta-programming, and is not prohibitive.

On the other hand, Fig. 16 shows the results for a pathological case where the size of the output program is exponential in the number of input processes. The source dataflow networks consists of N merge processes, N+1 input streams, and a single output stream. The output of each merge process is the input of the next, forming a chain of merges. In source notation the network for N = 3 is `sOut = merge sIn1 (merge sIn2 (merge sIn3 sIn4))`.

When fusing two processes the fusion algorithm essentially compares every state in the first process with every state in the second, computing a cross product. During the fusion transform, as states in the result process are generated they are added to a finite map — the `instrs` field of the process definition. The use of the finite map ensures that identical states are always combined, but genuinely different states always make it into the result.

In the worst case, fusion of two processes produces $O(n * m)$ different states, where n and m are the number of states in each. If we assume the two processes have about the same number of states then this is $O(n^2)$. Fusing the next process into this result yields $O(n^3)$, so overall the worst case number of states in the result will be $O(n^k)$, where k is the number of processes fused.

In the particular case of `merge`, the implementation has two occurrences of the `push` instruction. During fusion, the states for the consuming process are inlined at each occurrence of `push`. These states are legitimately different because at each occurence of `push` the input channels of the merge process are in different channel states, and these channel states are included in the overall process state.

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 Fig.5. At labels `F1` and `F2` are `jump` instructions, where the first passes control to the second. The update expressions attached to these instructions are also non-interferring, which means we can safely combine these consecutive `jump` instructions into a single `jump` to `F4`. 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 Fig.12 implement a heuristic that causes jump instructions to be scheduled before all others, so they tend to end up in these groups.

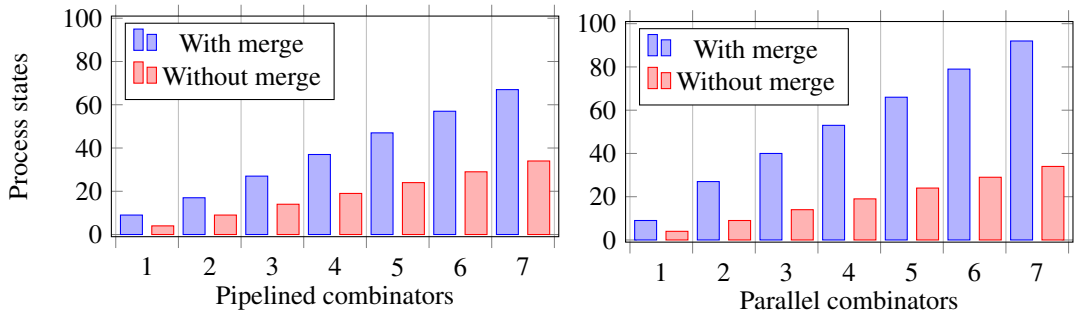

Fig. 15. Maximum output process size for fusing all combinations of up to *n* combinators.

There are also `jump` instructions like the one at `F5` which 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.

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

Another optimization is to notice that in some cases, when a heap variable is updated it is always assigned the value of another variable. In Fig.5, the `v` and `x1` variables are only ever assigned the value of `b1`, and `b1` itself is only ever loaded via a `pull` instruction. Remember from §2.3.1 that the variable `b1` is the stream buffer variable. Values pulled from stream `sIn1` are first stored in `b1` before being copied to `v` and `x1`. When the two processes to be fused share a common input stream, use of stream buffer variable allows one process to continue using the value that was last pulled from the stream, while the other moves onto the next one.

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 Fig.5, and we can perform a copy-propagation optimisation, replacing all occurrences of `v` and `x1` with the single variable `b1`. In the result, the interleaved instructions from both source processes then share the same heap variable.

To increase the chance that we can perform this above copy-propagation, we need both processess to want to pull from the same stream at the same time. In the definition of a particular process, 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. In general, the `drop` for a particlar stream should be placed just before a `pull` from the same stream.

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. Note that due to non-determinism of process execution the converse is not true in practice: just because the two concurrent processes can produce a particular output sequence does not mean the fused result process will as well — the fused result process uses only one of the many possible orders.

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

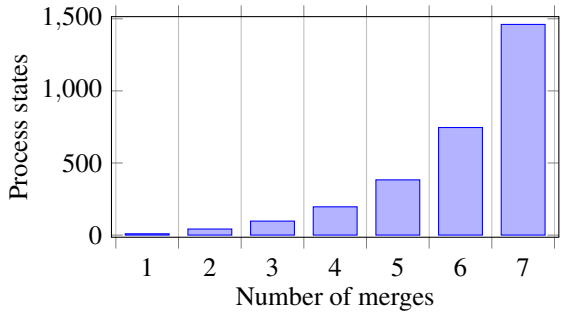

Fig. 16. Exponential blowup occurs when splitting or chaining merges together.

The `Soundness` theorem uses `EvalBs` to evaluate the fused program, and `EvalOriginal` ensures that the original program evaluates with that program's subset of the result heap, using `Var1` and `Var2` to extract the variables.

To aid mechanization, the Coq formalization has some small differences from the system presented in this paper. Firstly, the Coq formalization uses a separate `update` instruction to modify variables in the local heap, rather than attaching heap updates to the *Next* label of every instruction. Performing this desugaring makes the low level lemmas easier to prove, but we find attaching the updates to each instruction makes for an easier exposition. Secondly, our formalization only implements sequential evaluation for a single process, rather than non-deterministic evaluation for whole process groups as per Fig.9. Instead, 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`.

Short cut fusion is an attractive idea, as it allows fusion systems to be specified by a single rewrite rule. However, short cut fusion relies on inlining which, like pull-based streams, only occurs when there is a single consumer. Thus, short cut fusion is inherently biased towards pull fusion. Push-based short cut fusion systems *do* exist (Gill et al. 1993), but support neither `zip` nor `unzip` (Lippmeier et al. 2013; Svenningsson 2002).

Recent work on stream fusion by Kiselyov et al. (2017) uses staged computation in a push-based system to ensure all combinators are inlined, but when streams are used multiple times this causes excessive inlining, which duplicates work. For effectful inputs such as reading from the network, duplicating work changes the semantics.

Data flow fusion (Lippmeier et al. 2013) is neither pull-based nor push-based, and supports arbitrary splits and joins. It supports standard combinators such as `map`, `filter` and `fold`, and converts each stream to a series with explicit rate types, similar to the clock types of Lucid Synchrone (Benveniste et al. 2003). These rate types ensure that well-typed programs can be fused without introducing unbounded buffers. This allows unfusable programs to be caught at compile time. However, it only supports a limited set of combinators, and adding more is non-trivial.

One way to address the difference between pull and push streams is to explicitly support both separately, as seen in Bernardy and Svenningsson (2015) and Lippmeier et al. (2016). Here, pull streams have the type `Source` and represent a source that is always available to be pulled from, while push streams have the type `Sink` and represent a

sink that can always accept input. Both systems rely on stream bindings being used linearly to ensure correctness, including boundedness of buffers. Operations over sources are expressed fairly naturally compared to streams, for example the `zip` combinator has the type `Source a -> Source b -> Source (a,b)`. Sinks, however, are co-variant, and operations must be performed somewhat backwards, so that the `unzip` combinator takes the two output sinks to push into and returns a new sink that pushes into these. It has the type `Sink a -> Sink b -> Sink (a,b)`. This system requires the streaming computation to be manually split into sources and sinks, and be joined together by a loop that 'drains' values from the source and pushes them into the sink.

The duality between pull and push arrays has also been explored in Obsidian by (Claessen et al. 2012) and later in (Svensson and Svenningsson 2014). 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.

Streaming IO libraries have blossomed in the Haskell ecosystem, generally based on Iteratees (Kiselyov 2012). Libraries such as `conduit` (Snoyman 2011), `enumerator` (Millikin and Vorozhtsov 2011), `machines` (Kmett et al. 2012) and `pipes` (Gonzalez 2012) are all designed to write stream computations with bounded buffers. However, these libraries provide no fusion guarantees, and as such programs tend to be written over chunks of data to make up for the communication overhead. For the most part they support only straight-line computations, with only limited forms of branching.

In relation to process calculi, synchronised product has been suggested as a method for fusing Kahn process networks together (Fradet and Ha 2004), but does not appear to have been implemented or evaluated. The synchronised product of two processes allows either process to take independent or local steps at any time, but shared actions, such as when both processes communicate on the same channel, must be taken in both processes at the same time. This is a much simpler fusion method than ours, but is also much stricter. When two processes share multiple channels, synchronised product will fail unless both processes read the channels in exactly the same order. Our system can be seen as an extension of synchronised product that allows some leeway in when processes must take shared steps: they do not have to take shared steps at the same time, but if one process lags behind the other, it must catch up before the other one gets too far ahead.

Synchronous languages such as LUSTRE (Halbwachs et al. 1991), Lucy-n (Mandel et al. 2010) and SIGNAL (Le Guernic et al. 2003) all use some form of clock calculus and causality analysis to ensure that programs can be statically scheduled with bounded buffers. These languages describe *passive* processes where values are fed in to streams from outside environments, such as data coming from sensors. In this case, the passive process has no control over the rate of input coming in, and if they support multiple input streams, they must accept values from them in any order. In contrast, the processes we describe are *active* processes that have control over the input that is coming in. This is necessary for combinators such as mergesort-style `merge`, as well as `append`. Note that in the synchronous language literature, it is common to refer to a different merge operation, also known as `default`, which computes a stream that is defined whenever either input is defined.

Synchronous dataflow (not to be confused with synchronous languages above) is a dataflow graph model of computation where each dataflow actor has constant, statically known input and output rates. The main advantage of synchronous dataflow is that it is simple enough for static scheduling to be decidable, but this comes at a cost of expressivity. StreamIt (Thies et al. 2002) uses synchronous dataflow for scheduling when possible, otherwise falling back to dynamic scheduling (Soule et al. 2013). Boolean dataflow and integer dataflow (Buck 1994; Buck and Lee 1993) extend synchronous dataflow with boolean and integer valued control ports, and attempt to recover the structure of ifs and loops from select and switch actors. These systems allow some dynamic structures to be scheduled statically, but are very rigid and only support limited control flow structures: it is unclear how merge or append could be scheduled by this system. Finite state machine-based scenario aware dataflow (FSM-SADF) (Stuijk et al. 2011; Van Kampenhout et al. 2015) is still quite expressive compared to boolean and integer dataflow, while still ensuring static scheduling. A finite state machine is constructed, where each node of the FSM denotes its own

synchronous dataflow graph. The FSM transitions from one dataflow graph to another based on control outputs of the currently executing dataflow graph. For example, a filter is represented with two nodes in the FSM. The dataflow graph for the initial state executes the predicate, and the value of the predicate is used to determine which transition the FSM takes: either the predicate is false and the FSM stays where it is, or the predicate is true and moves to the next state. The dataflow graph for the next state emits the value, and moves back to the first state. This does appear to be able to express value-dependent operations such as merge, but lacks the composability - and familiarity - of combinators.

8 FUTURE WORK

8.1 Case analysis

The fusion algorithm treats all `case` conditions as fully abstract, by exploring all possible combinations for both processes. This can cause issues for processes that dynamically require only a bounded buffer, but the fusion algorithm statically tries every combination and wrongly asserts that unbounded buffering is required.

For example, suppose two processes have the same case condition (`x > 0`). The fusion algorithm will generate all four possibilities, including the contradictory ones such as where the first process has (`x > 0 = true`) and the second has (`x > 0 = false`). If any require an unbounded buffer, the fusion algorithm will fail.

A possible extension is to somehow cull these contradictory states so that if it is statically known that a state is unreachable, it does not matter if it requires unbounded buffers. It may be possible to achieve this with relatively little change to the fusion algorithm itself, by having it emit some kind of failure instruction rather than failing to produce a process. A separate postprocessing step could then perform analyses and remove statically unreachable process states. After postprocessing, if any failure instructions are reachable, fusion fails as before.

8.2 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 because it is simpler to explain and prove, but the extensions required to support finite streams do 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.

For a combinator such as `take` which only needs a finite prefix of the input stream, it may be useful to add an instruction that allows a pulling process to explicitly disconnect from an input channel. After disconnecting, the process no longer needs to be coordinated with the producer, leaving the producer free to push as often as the other consumers allow. We have implemented an early prototype that supports finite streams, our mechanized formalization does not cover it.

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 analyse and reason about case statements and their predicates. It seems obvious that if two consumers read the same value and check if it is EOF, they are both going to give the same result, but this is of course undecidable in general. By making the structure of finite streams explicit and constraining processes to use finite streams in particular ways, we may be able to give more guarantees than by relying on heuristics for deciding equality of predicates.

REFERENCES

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).

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: Implementation and Application of Functional Languages*.

Adnan Bouakaz. 2013. *Real-time scheduling of dataflow graphs*. Ph.D. Dissertation. Université Rennes 1.

Joseph T Buck. 1994. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *Signals, Systems and Computers, Twenty-Eighth Asilomar Conference on*.

Joseph Tobin Buck and Edward A Lee. 1993. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on*.

Koen Claessen, Mary Sheeran, and Joel Svensson. 2012. Expressive array constructs in an embedded GPU kernel programming language. In *DAMP: Declarative Aspects of Multicore Programming*.

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream fusion: From lists to streams to nothing at all. In *ACM SIGPLAN Notices*.

Paul Feautrier and Christian Lengauer. 2011. Polyhedron model. In *Encyclopedia of Parallel Computing*.

Pascal Fradet and Stéphane Hong Tuan Ha. 2004. Network fusion. In *Asian Symposium on Programming Languages and Systems*.

Andrew Gill, John Launchbury, and Simon L Peyton Jones. 1993. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*.

Gabriel Gonzalez. 2012. The pipes Haskell package. (2012). http://hackage.haskell.org/package/pipes.

Goetz Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge Data Engineering* 6, 1 (1994).

Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* (1991).

Gilles Kahn, David MacQueen, and others. 1976. Coroutines and networks of parallel processes. (1976).

Michael Kay. 2009. You pull, I'll push: on the polarity of pipelines. In *Balisage: The Markup Conference*.

Oleg Kiselyov. 2012. Iteratees. In *International Symposium on Functional and Logic Programming*.

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*.

Edward Kmett, Rnar Bjarnason, and Josh Cough. 2012. The machines Haskell package. (2012). http://hackage.haskell.org/package/machines.

Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. 2003. Polychrony for system design. *Journal of Circuits, Systems, and Computers* (2003).

Ben Lippmeier, Manuel MT Chakravarty, Gabriele Keller, and Amos Robinson. 2013. Data flow fusion with series expressions in Haskell. In *ACM SIGPLAN Notices*.

Ben Lippmeier, Manuel M. T. Chakravarty, Gabriele Keller, and Simon L. Peyton Jones. 2012. Guiding parallel array fusion with indexed types. In *Haskell Symposium*.

Ben Lippmeier, Fil Mackay, and Amos Robinson. 2016. Polarized data parallel data flow. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing*.

Louis Mandel, Florence Plateau, and Marc Pouzet. 2010. Lucy-n: a n-synchronous extension of Lustre. In *Mathematics of Program Construction*.

John Millikin and Mikhail Vorozhtsov. 2011. The enumerator Haskell package. (2011). http://hackage.haskell.org/package/enumerator.

Michael Snoyman. 2011. The conduit Haskell package. (2011). http://hackage.haskell.org/package/conduit.

Robert Soule, Michael I. Gordon, Saman Amarasinghe, Robert Grimm, and Martin Hirzel. 2013. Dynamic Expressivity with Static Optimization for Streaming Languages. In *The 7th ACM International Conference on Distributed Event-Based Systems*.

Sander Stuijk, Marc Geilen, Bart Theelen, and Twan Basten. 2011. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Embedded Computer Systems (SAMOS), International Conference on*.

Josef Svenningsson. 2002. Shortcut fusion for accumulating parameters & zip-like functions. In *ACM SIGPLAN Notices*.

Bo Joel Svensson and Josef Svenningsson. 2014. Defunctionalizing push arrays. In *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*.

William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *Compiler Construction*.

Reinier Van Kampenhout, Sander Stuijk, and Kees Goossens. 2015. A scenario-aware dataflow programming model. In *Digital System Design (DSD), Euromicro Conference on*.

A COMBINATORS

Here we show the definitions of some combinators. We start with simple combinators supported by most streaming systems, and progress to more interesting combinators. Some standard combinators such as `fold`, `take` and `append` are missing due to the infinite nature of our streams, but could be implemented with the finite stream extension described in §8.2. The fact that segmented versions of these combinators can be implemented is compelling evidence of this.

Many of these combinators take a "default" argument, which is used to initialise the heap, but the stored value is never actually read. Ideally these could be left unspecified, or the heap left uninitialised in cases where it is never read.

The map combinator applies a function to every element of the stream.

```
map
    = λ (f : α → β) (default : α) (sIn: Stream α) (sOut: Stream β).
      ν (a: α) (L0..L2: Label).
     process
     { ins:    { sIn }
     , outs:   { sOut }
     , heap:   { a = default }
     , label:  L0
     , instrs: { L0 = pull sIn     a  L1 []
               , L1 = push sOut (f a) L2 []
               , L2 = drop sIn        L0 [] } }
```

Filter returns a new stream containing only the elements that satisfy some predicate.

```
filter
    = λ (f : α → Bool) (default : α) (sIn: Stream α) (sOut: Stream α).
      ν (a: α) (L0..L3: Label).
     process
     { ins:    { sIn }
     , outs:   { sOut }
     , heap:   { a = default }
     , label:  L0
     , instrs: { L0 = pull sIn  a     L1 []
               , L1 = case   (f a)    L2 []  L3 []
               , L2 = push sOut a     L3 []
               , L3 = drop sIn        L0 [] } }
```

Partition is similar to filter, but has two output streams: those that satisfy the predicate, and those that do not. Partition is an inherently push-based operation, and cannot be supported by pull streams without buffering.

```
partition
    = λ (f : α → Bool) (default : α)
        (sIn: Stream α) (sOut1: Stream α) (sOut2: Stream α).
      ν (a: α) (L0..L4: Label).
     process
     { ins:    { sIn }
     , outs:   { sOut1, sOut2 }
     , heap:   { a = default }
     , label:  L0
     , instrs: { L0 = pull sIn   a    L1 []
               , L1 = case    (f a)   L2 []  L3 []
               , L2 = push sOut1 a    L4 []
               , L3 = push sOut2 a    L4 []
               , L4 = drop sIn        L0 [] } }
```

Zip, or zip-with, pairwise combines two input streams. Zipping is an inherently pull-based operation.

```
zipWith
    = λ (f : α → β → γ) (default1 : α) (default2 : β)
        (sIn1: Stream α) (sIn2: Stream β) (sOut: Stream γ).
      ν (a: α) (b : β) (L0..L4: Label).
     process
     { ins:    { sIn1, sIn2 }
     , outs:   { sOut }
     , heap:   { a = default1, b = default2 }
     , label:  L0
     , instrs: { L0 = pull sIn1 a       L1 []
               , L1 = pull sIn2 b       L2 []
               , L2 = push sOut (f a b) L3 []
               , L3 = drop sIn1         L4 []
               , L4 = drop sIn2         L0 [] } }
```

Scan is similar to a fold, but instead of returning a single value at the end, it returns an intermediate value for each element of the stream.

```
scan
    = λ (k : α → β → β) (z : β) (default : α) (sIn: Stream α) (sOut: Stream β).
      ν (a: α) (s : β) (L0..L2: Label).
     process
     { ins:    { sIn  }
     , outs:   { sOut }
     , heap:   { a = default, s = z }
     , label:  L0
     , instrs: { L0 = pull sIn  a       L1 []
               , L1 = push sOut s       L2 [ s = f a s ]
               , L2 = drop sIn          L0 [] } }
```

Segmented fold performs a fold over each nested stream, using a segmented representation. Here we are representing nested streams using one stream for the lengths of each substream, and another stream containing the values. The output stream has the same rate as the lengths stream. It reads a count (c) from the lengths stream, setting the fold state to zero (z). Then it reads count times from the values stream, updating the fold state. Afterwards, it pushes the final fold state, and continues to read a new count.

```
folds
    = λ (k : α → β → β) (z : β) (default : α)
        (sLens: Stream Nat) (sVals: Stream α) (sOut: Stream β).
      ν (c : Nat) (a: α) (s : β) (L0..L5: Label).
     process
     { ins:    { sLens, sVals }
     , outs:   { sOut }
     , heap:   { c = 0, a = default, s = z }
     , label:  L0
     , instrs: { L0 = pull sLens c      L1 [ s = z ]
               , L1 = case (c > 0)      L2 []  L4 []
               , L2 = pull sVals a      L3 []
               , L3 = drop sVals        L1 [ c = c - 1, s = k s a ]
               , L4 = push sOut  s      L5 []
               , L5 = drop sLens        L0 [] } }
```

Segmented take computes an n-length prefix of each nested stream. It starts by reading a count from the lengths stream, then copies at most n elements. If there are leftovers, it pulls and discards them, then pulls the next length.

```
takes
    = λ (n : Nat) (default : α)
        (sLens: Stream Nat) (sVals: Stream α)
        (oLens: Stream Nat) (oVals: Stream α).
      ν (c : Nat) (take : Nat) (ix : Nat) (a: α) (L0..L9: Label).
     process
     { ins:    { sLens, sVals }
     , outs:   { oLens, oVals }
     , heap:   { c = 0, take = 0, ix = 0, a = default }
     , label:  L0
     , instrs: { L0 = pull sLens c      L1 [ ix = 0, take = min count n ]
               , L1 = push oLens take   L2 []

               , L2 = case (ix < take)  L3 []  L6 []
               , L3 = pull sVals a      L4 []
               , L4 = push oVals a      L5 []
               , L5 = drop sVals        L2 [ ix = ix + 1 ]

               , L6 = case (ix < c)     L7 []  L9 []
               , L7 = pull sVals a      L8 []
               , L8 = drop sVals        L6 [ ix = ix + 1 ]

               , L9 = drop sLens        L0 [] } }
```

Segmented append takes two segmented streams as input, and appends each nested stream. It starts by reading a length from both lengths streams into a and b, and pushes the sum of both lengths. It then copies over a elements from the first values stream, then copies over b elements from the second values stream.

```
appends
    = λ (default : α)
        (aLens: Stream Nat) (aVals: Stream α) (bLens: Stream Nat) (bVals: Stream α)
        (oLens: Stream Nat) (oVals: Stream α).
      ν (a : Nat) (b : Nat) (v: α) (L0..L12: Label).
     process
     { ins:    { aLens, aVals, bLens, bVals }
     , outs:   { oLens, oVals }
     , heap:   { a = 0, b = 0, v = default }
     , label:  L0
     , instrs: { L0  = pull aLens  a     L1 []
               , L1  = pull bLens  b     L2 []
               , L2  = push oLens (a+b)  L3 []

               , L3  = case (a > 0)      L4 []  L7 []
               , L4  = pull aVals v      L5 []
               , L5  = push oVals v      L6 []
               , L6  = drop aVals        L3 [ a = a - 1 ]

               , L7  = case (b > 0)      L8 []  L11[]
               , L8  = pull bVals v      L9 []
               , L9  = push oVals v      L10[]
               , L10 = drop bVals        L7 [ b = b - 1 ]

               , L11 = drop aLens        L12[]
               , L12 = drop bLens        L0 [] } }
```