# Data Flow Fusion with Series Expressions in Haskell

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

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

## Abstract

Existing approaches to array fusion can deal with straight-line producer consumer pipelines, but cannot fuse branching data flows where a generated array is consumed by several different consumers. Branching data flows are common and natural to write, but a lack of fusion leads to the creation of an intermediate array at every branch point. We present a new array fusion system that handles branches, based on Waters's series expression framework, but extended to work in a functional setting. Our system also solves a related problem in stream fusion, namely the introduction of duplicate loop counters. We demonstrate speedup over existing fusion systems for several key examples.

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

***Keywords*** Arrays; Fusion; Haskell

## 1. Introduction

Consider the following `filterMax` function that increments a vector of integers and then selects just the positive results, while also determining the maximum value among the positive results.

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let vec2    = map    (+ 1) vec1
       vec3    = filter (> 0) vec2
       n       = fold max 0 vec3
   in  (vec3, n)
```

A client programmer would rightfully expect this code to be fused into a single loop, one that keeps track of the maximum value while it builds the result vector. Unfortunately, existing fusion methods, such as `build` / `fold` fusion [8] and stream fusion [7], cannot completely fuse this code. Existing methods can eliminate the construction of `vec2`, but cannot fuse the production of `vec3` into both of its consumers. As a result, the entire intermediate `vec3` will be materialized in memory by `filter`, before being read back by `fold`, requiring a second loop and superfluous memory traffic. If the input vector `vec1` is large, then `filterMax` is likely to be memory bound. For this reason, an incompletely fused `filterMax` will be about 50% slower than handwritten code that combines the `filter` and `fold` into a single loop.

Functions like `filterMax` are common. For example, the core operation of the QuickHull algorithm is a `filterMax`-like computation (§6). Branching data flows with multiple consumers are encountered whenever a function uses an array combinator that produces multiple result arrays, like with `unzip`, as the two results are typically fed into distinct consumers.

This problem of branching data flows is related to another problem with stream fusion, the standard in Haskell array fusion systems. Specifically, stream fusion of `zipWith`-like functions results in loops with duplicate loop counters, wasting precious processor registers and leading to redundant loop-counter arithmetic. This problem is especially severe in code produced by Data Parallel Haskell's vectorization transform [18] — we have seen loops with eight duplicate loop counters!

Our new fusion system solves these problems. Our main contributions are the following:

- We present a new fusion system, which we name *flow fusion*. Our new system is based on Waters's series expressions [27], extended to be useful in a functional setting. This system completely fuses functions like `filterMax` and avoids the duplicate loop counters that are common to stream fusion (§4).
- We show how to use rank-2 quantified rate type variables and rate conversion witnesses to satisfy the *online criteria* of the series expressions framework. Rate variables and the online criteria are related to the clock calculi used in synchronous dataflow languages (§3).
- We present key benchmarks showing performance improvement over existing array fusion systems. Properly fused QuickHull has a runtime half that of the stream fusion version, as its `filterMax`-like core requires only one array traversal instead of two (§6).

We have implemented flow fusion as a GHC optimization plugin that rewrites intermediate code generated by a user-facing library of Haskell source code. This Haskell library includes a fallback implementation of our array combinators, based on stream fusion, that is used if the plugin is not enabled. The code is available via `http://repa.ouroborus.net`.

## 2. The Problems with Stream Fusion

*Fusion*, or *deforestation*, refers to the automatic, compile time elimination of intermediate data structures, by combining successive traversals over these structures. Fusion has already received plenty

```
-- Vector versions
map    :: (a -> b) -> Vector a -> Vector b
map f xs    = unstream (mapS f (stream xs))

filter :: (a -> Bool) -> Vector a -> Vector a
filter f xs = unstream (filterS f (stream xs))

fold   :: (a -> b -> a) -> a -> Vector b -> a
fold f z xs = foldS f z (stream xs)

-- Stream versions
mapS     :: (a -> b)    -> Stream a -> Stream b
filterS  :: (a -> Bool) -> Stream a -> Stream a
foldS    :: (a -> b -> a) -> a -> Stream b -> a

-- Stream / Vector conversions
stream   :: Vector a -> Stream a
unstream :: Stream a -> Vector a
```

**Figure 1.** Stream Fusion Operators

of attention in the context of functional programming [7, 8, 11, 16, 20, 24, 27]. While most prior work aims to remove intermediate lists, some methods also apply to arrays [4, 6, 9, 13].

The most practically successful systems —`build/fold` fusion, stream fusion, and delayed arrays— are *short-cut fusion* methods that rely on local program transformations. These methods are implemented as simple but specific rewrite rules combined with general purpose program transformations. Systems like [11] and [9] also perform *tupling* that fuses restricted classes of branching data flows. Unfortunately, neither [11] or [9] handle `filterMax`, because the overall result includes a materialized intermediate array (`vec3`) as well as an additional value based on its elements (`n`). Tupling transformations handle the easier case where two results are computed by directly traversing over the same input structure.

In contrast, *loop fusion* methods used in imperative languages merge multiple loop nests, typically using dependency graphs [25] to determine whether fusion is legal and beneficial. When a producer and consumer loop are merged, array contraction [21] can then remove or reduce the size of the intermediate arrays. These systems require fusion-specific compiler support and more global reasoning than short-cut fusion. However, the simplicity of short-cut fusion comes at a price, as we discuss next.

### 2.1 Short-cut Fusion is Local and Depends on Inlining

Short cut fusion systems do not rely on custom program transformations that analyse entire functions or compilation units. Rather, they use inlining and let-floating to produce code in which array producers and consumers are adjacent. These producer-consumer pairs are then eliminated by rewrite rules and other local transformations. This approach permits a simple implementation, but limits the use of contextual information. To see why, consider the `filterMax` function again:

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let vec2    = map    (+ 1) vec1
       vec3    = filter (> 0) vec2
       n       = fold max 0 vec3
   in  (vec3, n)
```

The definitions of `map`, `filter` and `fold` for stream fusion [7] are shown in Figure 1. These are written in terms of co-recursive functions operating over streams, named `mapS`, `filterS`, and `foldS`, respectively — see [7] for details. The functions `stream` and `unstream` convert between the *vector-view* and the *stream-view* of the data, where we use "vector" to mean a one dimensional array. Inlining `map`, `filter`, `fold` into `filterMax` gives us:

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let vec3 = unstream
                  (filterS (> 0) (stream (unstream
                     (mapS (+ 1) (stream vec1)))))
       n    = foldS max 0 (stream vec3)
   in  (vec3, n)
```

Now we can use the following rewrite rule:

```
{-# RULE "stream/unstream"
         forall s. stream (unstream s) = s #-}
```

This rule says that if we convert a `Stream` to a `Vector` and back again, we get the original `Stream`. Applying this rule to our program yields:

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let vec3 = unstream (filterS (> 0)
                        (mapS (+ 1) (stream vec1)))
       n    = foldS max 0 (stream vec3)
   in  (vec3, n)
```

It is this application of the `stream/unstream` rule that actually eliminates the intermediate structure. General purpose transformations will then ensure that `filterS` and `mapS` are inlined, and their co-recursive definitions ensure no further intermediate structure will be allocated for the result of `mapS`.

Importantly, note that this mechanism does not apply to the `vec3` binding, because `vec3` has two consumers: being used by `foldS` and also returned in the final tuple. We cannot duplicate the `vec3` binding for each consumer as this would also duplicate the work required to produce its elements.

Short-cut fusion is fundamentally limited to data structures that have a single consumer. To fuse `vec3`, we must make use of non-local information to infer that the computation of `foldS` and `filterS` should be combined.

### 2.2 Short-cut Fusion Duplicates Loop Counters

Consider this simple expression, combining three vectors by adding two element-wise and then multiplying by the third:

```
zipWith (*) (zipWith (+) xs ys) zs
```

After inlining `zipWith`, using the `stream/unstream` rule, and then combining the two resulting instances of `zipWithS`, stream fusion produces the following code:

```
loop = \ i j k l s ->
 case >=# i len_xs of
  True  -> (# s, I# n #)
  False ->
   case indexIntArray# xs i of
    x -> case >=# j len_ys of
     True  -> (# s, I# n #)
     False ->
      case indexIntArray# ys j of
       y -> case >=# k len_zs of
        True -> (# s, I# n #)
        False ->
         case indexIntArray# zs k of
          z ->
           loop (+# i 1) (+# j 1) (+# k 1) (+# l 1)
            (writeIntArray# rs l (*# (+# x y) z) s)
```

We have four loop counters `i`, `j`, `k`, and `l` — three for the three source arrays and one for the result array. These counters contain

```
-- Haskell source after rate inference
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec
 = runSeries vec go1
 where go1 s1     = let s2    = map (+ 1) s1
                        flags = map (> 0) s2
                    in  mkSel flags (go2 s2)

       go2 s2 sel = let s4 = pack sel s2
                    in  ( create s4
                        , fold max 0 s4)

-- With explicit type abstraction and application
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec
 = runSeries @Int @(Vector Int, Int) vec go1
 where go1 = Λ(k1 : &). λ(s1 : Series k1 Int).
             let s2    : Series k1 Int
                       = map @k1 @Int @Int  (+ 1) s1
                 flags : Series k1 Bool
                       = map @k1 @Int @Bool (> 0) s2
             in  mkSel @k1 @(Vector Int, Int)
                       flags (go2 @k1 s2)

       go2 = Λ(k1 : &). λ(s2  : Series k1 Int).
             Λ(k2 : &). λ(sel : Sel k1 k2).
             let s4    : Series k2 Int
                       = pack @k1 @k2 @Int sel s2
             in  ( create @k2 @Int s4
                 , fold   @k2 @Int @Int max 0 s4)
```

**Figure 2.** The `filterMax` example after rate inference

```
runSeries  :: ∀(a b : *). Vector a
           -> (∀(k : &). Series k a -> b) -> b

runSeries2 :: ∀(a b c : *). Vector a -> Vector b
           -> (∀(k : &). Series k a -> Series k b -> c)
           -> Maybe c

generate   :: ∀(a : *). Int -> (Int -> a)
           -> (∀(k : &). Series k a -> b) -> b

create     :: ∀(k : &). ∀(a : *)
           .  Series k a -> Vector a

map        :: ∀(k : &). ∀(a b : *)
           .  (a -> b)
           -> Series k a -> Series k b

map2       :: ∀(k : &). ∀(a b c : *)
           .  (a -> b -> c)
           -> Series k a -> Series k b -> Series k c

fold       :: ∀(k : &). ∀(a b : *)
           .  (a -> b -> a) -> a -> Series k b -> a

mkSel      :: ∀(k1 : &). ∀(a : *)
           .  Series k1 Bool
           -> (∀(k2 : &). Sel k1 k2 -> a)   -> a

pack       :: ∀(k1 k2 : &). ∀(a : *)
           .  Sel k1 k2 -> Series k1 a -> Series k2 a
```

**Figure 3.** Series Operators

the same value, are incremented in lock step, and three of them are tested for loop bounds. In addition to the superfluous tests and arithmetic, the duplication of counters unnecessarily increases register pressure. Instead of hoping that subsequent optimizations will eliminate the duplicates (which is not done in the current version of GHC), flow fusion avoids their introduction entirely.

## 3. Rates and Contexts

The first definition in Figure 2 shows the code of `filterMax` after performing *rate inference* as a pre-processing step. Rate inference identifies regions of code amenable to flow fusion. Moreover, it decomposes rate changing operations such as `filter`, into primitives, such as `mkSel` and `pack`. As part of this decomposition, we have also introduced two intermediate `go` bindings to clarify data dependencies. We will explain rate inference in §3.6, but first we discuss the importance of rates for flow fusion.

The second definition in Figure 2 is the same function as before, but with explicit type abstractions and applications. Here Λ indicates type abstraction and `@` type application. We use `&` as the *kind of rate types*.

### 3.1 Vectors and Series

Rate inference ensures that subexpressions subjected to flow fusion contain only the array combinators shown in Figure 3. In those signatures, `Vector a` is the type of *manifest linear arrays*, represented by contiguous blocks of machine words in memory. In contrast, `Series k a` is the type of *abstract representations of sequences* of `a`-values produced at rate `k`, such that they may participate in fusion. In this respect a series is similar to a *delayed array* from Repa [13], except that series do not support random access indexing.

The *rate* `k` of a series is a type level representation of its length, with the following key invariant: *two series of the same rate are guaranteed to have the same length*. We use `&` as the kind of rate types, leaving `*` as the kind of value types as in standard Haskell. This distinction is important because rates are purely type-level information. There are no values that have a type of kind `&`.

The rate of a series is similar to a *clock type*, as used by the clock typing systems of synchronous data flow languages such as Lustre [1] and Lucid Synchrone [3, 19]. We compare these systems further in §7

### 3.2 Running Series Expressions

Rate inference encapsulates regions of fusible code in an abstraction `fn`, that is then passed as an argument to the function `runSeries`, `runSeries2`, or `generate`. The types for these functions are in Figure 3. The application `(runSeries v fn)` converts the vector `v` to a `Series`, which is then consumed by `fn`. In the type of `runSeries`, the rate variable `k` (which is a type-level representation of the length of `v`) is universally quantified in the type of `fn`. The inner quantification ensures that the rate information cannot escape, and that multiple series of differing lengths can never have the same rate variable. This is much like the use of a rank-2 type to encapsulate state in the `runST` function of the `ST` monad [14].

The inner quantification of a rate variable logically creates a *rate context*, where array data is processed at the specified rate. For example, the body of `go1` in Figure 2 is a rate context in which we construct two new `Series` values, `s2` and `flags`, both with the same rate `k1` derived from the incoming series `s1`.

Flow fusion can merge all computations producing and consuming series at the same rate into a single loop. For example, fused code that performs multiple folds over multiple series at the same rate will proceed as follows: first, the code initializes an accumulator for each fold; then, in a single loop, it reads elements from each series in lock step and updates the accumulators appropriately.

The `runSeries2` function is like `runSeries`, but accepts two `Vectors` *of the same length* as inputs and converts them both to `Series` before passing them to the worker. If the input vectors do

not have the same length then `runSeries2` yields `Nothing`. This dynamic check justifies our use of the same rate variable `k` for both `Series`. The `generate` function is like `runSeries`, but generating a series from a function producing array elements, instead of from a manifest vector.

Finally, `create` materializes a `Series` into a `Vector`. The latter is free of an associated rate variable, and hence can be passed outside the current rate context.

### 3.3 Maps and Folds

The types of `map` and `fold` in Figure 3 are standard, except for the inclusion of the rate variable `k`. The function `map2` is much like Haskell's standard `zipWith`, but requires both series to have the same rate (length). In our full implementation we have an entire family of functions `map3`, `map4` and so on.

In the explicitly typed code of Figure 2, the rate variables used as type arguments identify which rate context each operator belongs to. We will see in §4.1.2 that these type arguments indicate the set of series that ought to be evaluated in a single loop.

### 3.4 Selectors and Packing

In the expression `(pack sel s2)`, the *selector* `sel` identifies the values in series `s2` that should be included in the shorter result series. For example, suppose that during evaluation of `go2` from Figure 2, we have the following values for `s2` and `flags`, where `flags` identifies the positive elements of `s2`:

```
s2    :: Series k1 Int
s2    = [+4 -1 +5 +3 +8 -4 +2 +1 -5]

flags :: Series k1 Bool
flags = [ T  F  T  T  T  F  T  T  F]
```

The application `(mkSel flags fn)` converts `flags` into a *selector*, which is passed to `fn`, just as with `runSeries` before. The selector is an abstract representation of the vector containing the indices of all the `T` (True) values in that series:

```
sel :: Sel k1 k2
sel = [0 2 3 4 6 7]
```

The selector is then used by `pack` to select just those elements of `s2` that had their corresponding flag set:

```
s4  :: Series k2 Int
s4  = pack sel s2  =  [+4 +5 +3 +8 +2 +1]
```

Importantly, because the selector maps a series of one length onto a series of another, we tag the selector type with two different rate variables. For `filterMax`, we have `sel :: Sel k1 k2` where `k1` is the rate of `s2` and `flags`, and `k2` is the rate of `s4` — the series resulting from `pack`. Additionally, because selectors are always produced from a series of flags, we know that the length (rate) of the selector is no greater than the length (rate) of the original series. To put this another way, the rate context `k2` is *contained by* the rate context `k1`, and the selector is *evidence* and a *witness* for this relationship. Figure 5 shows this graphically.

### 3.5 Data Flow Languages and Clock Calculi

Figure 5 shows that `filterMax` is a first order, non-recursive dataflow program, as one might expect. The expressions that flow fusion can fuse all have this form: they consist of a number of manifest data sources, and a hierarchy of well nested rate contexts containing a directed acyclic graph of data flow operators terminated by manifest data sinks. Any program of this form can be completely fused by flow fusion.

The programs that we handle constitute a fragment of a more general data flow language such as Lustre [1] or Lucid Synchrone [19]. These larger languages work over infinite streams with recursion and delay elements, and prior work on compiling them has focused on dealing with these extra features [10]. These languages come with clock typing systems that ensure the program can be evaluated synchronously, and without unbounded buffering. In contrast, the fragment that we compile is defined by the API of Figure 3, which only provides finite series. We do not have delay elements or recursion. We use rate variables to express relationships between different series, but as we start with a simpler language, we can get by with a simpler rate analysis as described next.

### 3.6 Rate Inference

Rate inference identifies non-recursive data flow expressions that are amenable to flow fusion, and turns them into applications of `runSeries`. These expressions may only contain the following operators: (1) the vector operators from Figure 3 and (2) operators with scalar argument and result types. All higher-level array operators must be implemented in terms of these primitives to participate in flow fusion. Before rate inference, we assume the definitions of these composite operators have been inlined and the resulting code converted to a-normal form (administrative-normal form).

Rate inference proceeds in two phases: first, we identify, and if necessary, rearrange vector-valued subexpressions that we can fuse into single loops. For this we adapt the existing *size inference and scheduling algorithm* described by Chatterjee et al [5].

After solving the constraints as in [5], we proceed to the second phase. We rewrite the expression using operators on `Vectors` to use operators on `Series`, and wrap it in a `runSeries`. For this we replace all free vector-valued variables $\mathtt{v}_1\mathtt{::Vector}\ \mathtt{a}_1$ to $\mathtt{v}_n\mathtt{::Vector}\ \mathtt{a}_n$ with fresh variables $\mathtt{s::Series}\ k\ \mathtt{a}_i$. The rate inference algorithm ensures that all such variables have the same rate `k`. Rewriting the expression to use series operators is mostly trivial. Only `filter` needs special handling, to expand each occurrence into a use of `mkSel` and `pack`. As the input code is already in ANF, `filter` can only occur in bindings of the form:

```
let s1 = filter fn s0
in  e2
```

which we rewrite to

```
let flags = map fn s0
in  mkSel flags (λsel. let s1 = pack sel s0 in e2)
```

Finally, we wrap the series expression `e'` obtained this way with a `runSeries`:

```
runSeriesN v_1 ... v_n (λ s_1 ... s_n. e')
```

## 4. Loop Generation

After rate inference, our compilation method is as follows:

1. Type check and desugar Haskell source code to GHC Core.
2. A-normalize and eta-expand intermediate code.
3. *Slurp* out a data flow graph for each series process.
4. *Schedule* the operators in this graph into an abstract loop nest.
5. *Concretize* rate variables into loop counters.
6. *Extract* new Core code from the loop nest.
7. Inline library functions into the extracted code.
8. Complete compilation with GHC's standard pipeline.

A *series process* is a computation that can be expressed as a static, first order, non-recursive data flow graph like that of Figure 5. Data flow graphs are represented by the `Process` language shown in Figure 4. Abstract loop nests are represented by the `Procedure`

*name* → (process name)
$x$, $s$ → (value variable)
$a$, $k$ → (type variable)

*kind* ::= `*` | `&`

*type* ::= $a$ | `Int` | `Float` | ...

*process*
::= `process` *name* $(k_{in} : \&)$ $\overline{(a : kind)}$ $\overline{(x : type)}$ $\overline{(s : \texttt{Series}\ k_{in}\ type)}$
`with` $\overline{operator}$ `yields` *exp*

*operator*
::= `mkSel` $(k_{inner} : \&)$ $(x_{sel} : \texttt{Sel}\ k_{outer}\ k_{inner})$
`from` $k_{outer}$ $s_{flags}$ `in` $\overline{operator}$
| $s_{out}$ `<-` `map`$^n$ $k_{in}$ $\overline{type}^n$ $exp_{work}$ $\overline{s_{in}}^n$
| $s_{out}$ `<-` `pack` $k_{out}$ $k_{in}$ $type_{in}$ $x_{sel}$ $s_{in}$
| $x_{result}$ `<=` `fold` $k_{in}$ $type_{in}$ $type_{result}$ $s_{in}$ `with` $exp_{work}$ `and` $exp_{zero}$
| $x_{vec}$ `<=` `create` $k_{in}$ $type_{in}$ $s_{in}$

*exp* ::= ... Haskell expressions ...

**Figure 4.** Data Flow Process Description

language of Figure 6. In our current implementation stages 1, 7 and 8 are performed by GHC proper using its internal Core language, while stages 2-6 are performed by our GHC plugin. Note that the *Schedule* phase (described in §4.2) is really the core of our method, with the other phases performing impedance matching between the input and output languages.

### 4.1 Slurping Processes

The Slurp phase takes a normalized Core module and produces a list of fusible series processes.

```
slurp  :: Module -> [Process]
```

We supply the Core version of each series process to be fused as a top-level binding in the `Module`. During normalization (stage 2) the application of `runSeries` that creates the outer-most rate context is also split from the rest of the input code and floated to top-level. The `runSeries` function itself is implemented in an external Haskell library, and is not part of the Core program given to the loop generator. For our `filterMax` example, we would then have a binding of the following type:

```
filterMax_series
 :: ∀(k : &). Series k Int -> (Vector Int, Int)
```

This `filterMax_series` function is the same as `go1` from Figure 2, after it has been floated to top-level.

The `Process` language represents the data flow graph for a series process directly, without admitting language features that may be supported by the source language (Haskell) but not representable as a static, first-order data flow graph. If the Core version of the series process cannot be converted to our internal `Process` language, then the user gets a compile-time warning and the program is compiled via the fallback library discussed in §6. An example of this is in §4.1.2. On the other hand, if we *can* convert the source program to a `Process`, then we guarantee it will be completely fused.

The grammar for `Process` is shown in Figure 4, and here is the process description for our `filterMax` example which encodes the graph in Figure 5:

```
process filterMax_s (k1 : &) (s1 : Series k1 Int)
 with { s2 <- map k1 Int (+ 1) s1
      ; s3 <- map k1 Int (> 0) s2
      ; mkSel (k2 : Rate) (sel : Sel k1 k2)
          from  k1 s3  in
        { s4   <- pack   k1 k2 Int sel s2
        ; vec' <= create k2 Int s4
        ; mx   <= fold k2 Int s4 with (+) and 0 } }
 yields (vec', mx)
```

A *name* is a process name like `filterMax_s` (where `_s` indicates the `Process` version of this function). We use $x$ and $s$ as (meta) value variables, and by convention use $s$ for series and $x$ for non-series variables. We use $a$ and $k$ as (meta) type variables, where $a$ indicates an element type variable and $k$ indicates a rate.

We use *type* for element types, with the full set being defined by the host language (Haskell). Our program transformations treat element types abstractly.

A *process* consists of its name, its type and value parameters, a list of series operators, and an expression that yields the result of the overall process. We have left *exp* unspecified as this represents expressions from the source language — Haskell in our case. The rates of all input series must be identical to the first type variable $k_{in}$. This is the rate of the loop that we will generate for this process.

An *operator* can introduce a new rate context (`mkSel`), be a transformation that converts some series into another one (`map` and `pack`), or a *sink* that consumes some series and produces a non-series result (`fold` and `create`). Our operators are explicitly typed, being applied to rate variables that describe the context of each operator, and type arguments that give the element types of each series. Each operator defines a node in the data flow graph, where the binding symbols `<-` and `<=` represent the edges. The bindings in an operator list are non-recursive, and variables must be bound before they are used.

The `mkSel` construct binds the new variables $k_{inner}$ and $x_{sel}$ which are added to the environment of the enclosed list of operators. The `mkSel` operator itself defines a new rate context $k_{inner}$, inside an outer context $k_{outer}$. It consumes a series of flags $s_{flags}$ and produces a selector $x_{sel}$. In the enclosed operator list, all new series bound at that level must have rate $k_{inner}$. In Figure 5 we have drawn `mkSel` over the dotted line separating the rate contexts `k1` and `k2` to indicate that this operator defines the inner context.

The `map`$^n$ operator combines several input series $\overline{s_{in}}$ at rate $k_{in}$ using the worker function defined by $exp_{work}$. The $n$ variable sets the number of input series, though we write just `map` for `map1`.

The `pack` operator takes a series $s_{in}$ of rate $k_{in}$ to a series $s_{out}$ of rate $k_{out}$ using the selector $x_{sel}$.

The `fold` operator binds a new variable $x_{result}$ of $type_{result}$ which is the result of reducing the elements of series $s_{in}$ at rate $k_{in}$ using worker function $exp_{work}$ and neutral element defined by $exp_{zero}$. The $type_{in}$ argument is the type of the elements.

The `create` operator binds a new variable $x_{vec}$ which is the vector created from elements of the series $s_{in}$, at rate $k_{in}$ and element type $type_{in}$. In Figure 5 we have drawn the results produced by `fold` and `create` in square boxes to indicate that these are manifest values and not fusible series.

#### 4.1.1 Scoping in Series Process Descriptions

In a process description, the series that are parameters to the process, as well as the new series bound by each operator (to the left of the `<-` marks) can only be used as series arguments to other operators. These new series are *not in scope* in the worker expressions

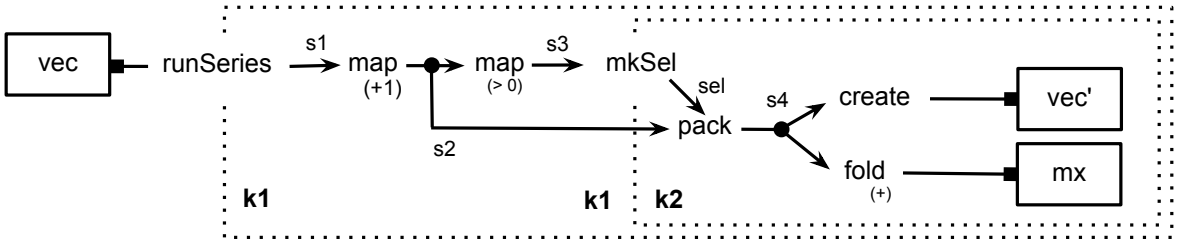


**Figure 5.** Nested Rate Contexts for `filterMax`

($exp_{work}$) and cannot be used in the expression given to `yields`. The results produced by series *sinks* (to the left of the `<=` marks) are also not in scope of the workers, but can be used in the expression given to `yields`. To put this another way, the workers may refer to environment variables defined outside the process they are contained in, but not variables bound internally in the process. These rules are needed to reject processes like the following:

```
process badNorm (k1 : &) (s1 : Series k1 Float)
 with { total <= fold k1 Float s1 with (+) and 0
      ; s2    <- map  k1 Int  (/ total) s1
      ; vec   <= create k1 Int s2 }
 yields vec
```

Here, because the worker function for `map` refers to the result `total`, to evaluate this code we would need to finish computation of the `fold` operator before embarking on the `map`. However, as we will see in §4.2, we intend to compile each process description into a single fused loop, and this is not possible for `badNorm`. It would be possible to automatically split such *compound* processes into individual parts before passing them to the scheduler defined in §4.2, but we leave this to future work.

#### 4.1.2 Normalizing the Input Core Program

The Core version of a series process is easy to slurp to the `Process` language of Figure 4, provided we make use of the explicit type annotations in Core, and perform some appropriate normalizations beforehand.

Before conversion, we a-normalize and eta-expand the definitions in the module so that every intermediate series has an explicit name. These names are identified with the edges of the extracted data-flow graph. We also use a preparation transform to force worker functions to be floated into their use sites — so that combinators like `mkSel`, `map` and `fold` are directly applied to workers. For example, as part of the `filterMax` example we get the following Core snippet:

```
mkSel @k1 @(Vector Int, Int) s1
  (Λ(k2 : &). λ(sel : Sel k1 k2).
   let s4 : Series k2 Int
          = pack @k1 @k2 @Int sel s2
   in ...)
```

With the above code we already have all the information we need to produce the equivalent snippet of the `Process` language:

```
mkSel (k2 : &) (sel : Sel k1 k2)
     from k1 s1 in
  { s4  <= pack k1 k2 Int sel s2
    ... }
```

Although the example above is really just a change of syntax, as mentioned in §4.1 the real point is that the `Process` language is smaller than the input Core language. We use the intermediate `Process` language primarily as way to reject features of the input language that cannot be expressed as static, first order, non recursive data flow graphs. For example, the following function cannot be converted to a `Process` because we have no way to represent the inner `if` construct.

```
badSwitchy :: ∀(k : &). Bool
           -> Series k Int -> Series k Int -> Int
badSwitchy flag s1 s2
 = fold (+) 0 (if flag then (map (* 2) s1)
                       else (map (* 4) s2)
```

The above function does not express a *static* data flow graph, because we do not statically know which input series to use for the `fold` operator. We have no way to compile this function into a single loop that fuses the contained `fold` and `map` operators, even though they operate on series all at the same rate.

#### 4.1.3 Processes are Hyper-strict

A process description is naturally hyper-strict, meaning every value that is mentioned will be computed. If we are not careful then this can lead to unused values being computed. For example, consider the following function that choses between two fold results:

```
strictSwitchy :: ∀(k : &). Bool
              -> Series k Int -> Series k Int -> Int
strictSwitchy flag s1 s2
 = choose flag (fold (+) 0 (map (* 2) s1))
               (fold (+) 0 (map (* 4) s2))
```

The above function is similar to `badSwitchy` from the previous section, except that we have used the `if`-like function `choose` to select between the two folded results. When evaluated using Haskell semantics, the fact that `choose` is non-strict in both arguments will mean that only one of the `fold` results will be computed. However, a-normalizing and then slurping a `Process` from this code produces:

```
process strictSwitchy (k : &) (flag : Bool)
         (s1 : Series k Int) (s2 : Series k Int)
 with { s3 <- map  k Int (* 2) s1
      ; s4 <- map  k Int (* 4) s2
      ; x1 <= fold k Int s3 with (+) and 0
      ; x2 <= fold k Int s4 with (+) and 0 }
 yields (choose flag x1 x2)
```

As mentioned earlier, we intend to compile this whole process description into a single loop. If we use the code above then both reductions will be computed at the same time, before choosing the desired result after the loop completes.

To avoid this problem automatically we could check whether values produced with `fold` or `create` are used strictly in the expression given to `yields`, and if not, emit a warning to the user.

$$
\begin{array}{lll}
\textit{name} & \rightarrow & \text{(procedure name)} \\
x,\ s & \rightarrow & \text{(value variable)} \\
a,\ k & \rightarrow & \text{(type variable)} \\
\textit{vec} & \rightarrow & \text{(vector variable)} \\
\textit{acc} & \rightarrow & \text{(accumulator)}
\end{array}
$$

$$\textit{kind} \quad ::= \texttt{*} \mid \texttt{\&}$$

$$\textit{type} \quad ::= a \mid \texttt{Int} \mid \texttt{Float} \mid \ldots$$

$$
\begin{array}{lll}
\textit{procedure} & ::= & \texttt{procedure}\ \textit{name}\ (k_{in} : \&)\ \overline{(a : \textit{kind})} \\
& & \quad \overline{(x : \textit{type})}\ \ \overline{(s : \texttt{Series}\ k_{in}\ \textit{type})} \\
& & \quad \texttt{with}\ \overline{\textit{nest}}\ \texttt{yields}\ \textit{exp}
\end{array}
$$

$$
\begin{array}{lll}
\textit{nest} & ::= & \texttt{loop}\ k \\
& & \quad \texttt{start}\ \ \overline{\textit{stmtstart}} \\
& & \quad \texttt{body}\ \ \overline{\textit{stmtbody}} \\
& & \quad \texttt{inner}\ \ \overline{\textit{nest}} \\
& & \quad \texttt{end}\ \ \overline{\textit{stmtend}} \\
& \mid & \texttt{guard}\ (k_{inner} : \&)\ \texttt{with}\ k_{outer}\ x_{flag} \\
& & \quad \texttt{body}\ \ \overline{\textit{stmtbody}} \\
& & \quad \texttt{inner}\ \ \overline{\textit{stmtend}}
\end{array}
$$

$$
\begin{array}{lll}
\textit{stmtstart} & ::= & \textit{vec}\ \ = \texttt{newVec}\ k \\
& \mid & \textit{acc}\ \ = \texttt{newAcc}\ \textit{exp}_{zero}
\end{array}
$$

$$
\begin{array}{lll}
\textit{stmtbody} & ::= & x_{elem}\ \ = x'_{elem} \\
& \mid & x_{elem}\ \ = \texttt{next}\ k\ s_{in} \\
& \mid & x_{elem}\ \ = \textit{exp}_{worker}\ \overline{x_{elem}} \\
& \mid & \textit{acc}\ \ := \textit{exp}_{worker}\ \textit{acc}\ x_{elem} \\
& \mid & \texttt{writeVec}\ k\ \textit{vec}\ x_{elem}
\end{array}
$$

$$
\begin{array}{lll}
\textit{stmtend} & ::= & x_{result}\ \ = \texttt{read}\ \textit{acc} \\
& \mid & \texttt{sliceVec}\ k\ \textit{vec}
\end{array}
$$

**Figure 6.** Series Procedures as Abstract Loop Nests

It may also be possible to automatically massage the source program or `Process` descriptions to ensure that unused results are not computed, but we have not investigated this further. As mentioned in §1, a primary client of our new fusion system is Data Parallel Haskell (DPH), and the DPH vectorizer eliminates conditional operators like `choose` as part of its existing vectorization process.

### 4.2 Scheduling Processes into Procedures

The Schedule phase takes a list of processes and converts it to a list of procedures.

```
schedule    :: [Process] -> [Procedure]
```

The definition of `Procedure` is given in Figure 6. A `Procedure` expresses the same computation as a `Process`, except that it is defined by an abstract, imperative loop nest instead of an operator graph. In Figure 6 the fields of the `loop` construct represent the *anatomy* of the loop, similarly to [22]. The idea of representing a loop in this "broken up" format also appears in work by Waters on series expressions [26, 27], which was inspired by the `loop` macro package of Common LISP [23].

For example, here is a simple `Process` that sums the elements of an input series `s1`:

```
process sum_s (k : &) (s1 : Series k Int)
 with { x1 <= fold k Int s1 with (+) and 0 }
 yields x1
```

The scheduled procedure is then:

```
procedure sum_d (k : &) (s1 : Series k Int)
 with loop k
      start { x1_acc   = newAcc 0 }
      body  { s1_elem  = next k s1
            ; x1_acc  := (+) x1_acc s1_elem }
      inner {}
      end   { x1       = read x1_acc }
 yields x1
```

The `start` field of a `loop` holds setup statements to execute before entering the loop proper; `body` contains the statements to execute for each iteration; `inner` holds some inner nests to run for every iteration, and `end` contains cleanup code that runs after the loop has completed. In future we elide empty fields, instead of writing `inner {}` and so on.

In the `sum_d` example, `x1_acc = newAcc 0` creates a new accumulator `x1_acc`. The statement `s1_elem = next k s1` takes the next element from `s1`. The statement `x1 = read x1_acc` reads the accumulator. Note that the `next`, update `(:=)` and `read` statements are side effecting, imperative operations.

Now, suppose we wanted a procedure like `sum_d` that also computed the product of `s1` concurrently with its sum, then added both results together. We would do this by appending an extra statement to each of the fields of the nest, and changing the yielded expression.

```
procedure sumProd (k : &) (s1 : Series k Int)
 with loop k
      start { acc1      = newAcc 0
            ; acc2      = newAcc 1 }          *NEW
      body  { s1_elem   = next k s1
            ; acc1     := (+) acc1 s1_elem
            ; acc2     := (*) acc2 s1_elem }  *NEW
      end   { x1        = read acc1
            ; x2        = read acc2 }         *NEW
 yields (x1 + x2)                           *CHANGED
```

In general terms, to convert a process into a procedure we consider each operator from the process in turn, and insert statements into the procedure that implement that operator. The fact that our procedure is expressed as an *anatomy* of separate `start`, `body` and `end` fields means that we can produce code that interleaves the computation of each operator. This interleaving of code is the primary mechanism which lets us deal with branching data flows, which we will return to in §7.

The scheduling process is formalized in Figure 7, and we will give more examples in coming sections. The top level judgment *process* ⇒ *procedure* converts a process into a similarly named procedure. Note that the resulting procedure has the same parameters and yielded expression, but its implementation has changed from an operator list into an abstract loop nest.

The ▷ operator used in Figure 7 has the following type:

$$\triangleright : \textit{nest} \rightarrow ((\textit{rate} + \top) \times \textit{field}) \rightarrow \textit{nest}$$

where *field* is one of the fields attached to the `loop` or `guard` constructs of Figure 6 (`start`, `body` and so on). The application $(n \triangleright k \times f)$ recursively descends into the nest $n$ until it finds the `loop` or `guard` construct that matches rate $k$, and then appends the statements in field $f$ to the similar field of that construct. When searching for matching rates we use the $k$ in (`loop` $k$) and the $k_{inner}$ in (`guard` $(k_{inner} : \&)$ `with` ... ). The `guard` construct is an abstract `if` expression, and $k_{inner}$ corresponds to the number of times the body is entered. In some rules we use $\top$ in place of a real rate variable, which matches the rate of the outer-most loop construct in the nest.

$\boxed{process \Rightarrow procedure}$

$$\frac{\texttt{loop}\ k_{in}\ \texttt{body}\ \{\overline{s_n^{elem} = \texttt{next}\ k_{in}\ s_n}^{n}\} \vdash \overline{operator} \Rightarrow nest}{\begin{array}{l}\texttt{process}\ \ name\ (k_{in} : \&)\ \overline{(a : kind)}\ \overline{(x : type)}\ \overline{(s_n : \texttt{Series}\ k_{in}\ type_n)}^{n}\ \texttt{with}\ \overline{operator}\ \texttt{yields}\ exp \\ \Rightarrow\ \texttt{procedure}\ name\ (k_{in} : \&)\ \overline{(a : kind)}\ \overline{(x : type)}\ \overline{(s_n : \texttt{Series}\ k_{in}\ type_n)}^{n}\ \texttt{with}\ nest\ \ \texttt{yields}\ exp\end{array}}$$

$\boxed{nest \vdash \overline{operator} \Rightarrow nest}$

$$\frac{nest \rhd (k_{outer} \times \texttt{inner}\ \{\ \texttt{guard}\ (k_{inner} : \&)\ \texttt{with}\ k_{outer}\ s_{flags}^{elem}\ \}) \vdash \overline{operator} \Rightarrow nest'}{nest \vdash \texttt{mkSel}\ (k_{inner} : \&)\ (x_{sel} : \texttt{Sel}\ k_{outer}\ k_{inner})\ \texttt{from}\ k_{outer}\ s_{flags}\ \texttt{in}\ \overline{operator} \Rightarrow nest'}$$

$$nest \vdash s_{out}\ \texttt{<-}\ \texttt{map}^n\ k_{in}\ \overline{type}^n\ exp_{work}\ \overline{s_{in}}^n \Rightarrow nest \rhd (k_{in} \times \texttt{body}\ \{\ s_{out}^{elem}\ \texttt{=}\ exp_{work}\ \overline{s_{in}^{elem}}^n\})$$

$$nest \vdash s_{out}\ \texttt{<-}\ \texttt{pack}\ k_{out}\ k_{in}\ type_{in}\ x_{sel}\ s_{in} \Rightarrow nest \rhd (k_{out} \times \texttt{body}\ \{\ s_{out}^{elem}\ \texttt{=}\ s_{in}^{elem}\ \})$$

$$\begin{array}{lll} nest \vdash x_{result}\ \texttt{<=}\ \texttt{fold}\ k_{in}\ type_{in}\ type_{result}\ s_{in} & \Rightarrow nest \rhd (\top & \times\ \texttt{start}\ \{\ x_{result}^{acc}\ \texttt{=}\ \texttt{newAcc}\ exp_{zero}\ \}) \\ \qquad \texttt{with}\ exp_{work}\ \texttt{and}\ exp_{zero} & \rhd (k_{in} & \times\ \texttt{body}\ \{\ x_{result}^{acc}\ \texttt{:=}\ exp_{work}\ x_{result}^{acc}\ s_{in}^{elem}\ \}) \\ & \rhd (\top & \times\ \texttt{end}\ \{\ x_{result}\ \texttt{=}\ \texttt{read}\ x_{result}^{acc}\ \}) \\ nest \vdash x_{vec}\ \texttt{<=}\ \texttt{create}\ k_{in}\ type_{in}\ s_{in} & \Rightarrow nest \rhd (\top & \times\ \texttt{start}\ \{\ x_{vec}\ \texttt{=}\ \texttt{newVec}\ k_{in}\ \}) \\ & \rhd (k_{in} & \times\ \texttt{body}\ \{\ \texttt{writeVec}\ k_{in}\ x_{vec}\ s_{in}^{elem}\ \}) \\ & \rhd (\top & \times\ \texttt{end}\ \{\ \texttt{sliceVec}\ k_{in}\ x_{vec}\ \}) \end{array}$$

**Figure 7.** Scheduling Series Processes into Procedures

#### 4.2.1 Scheduling Maps

As an example of how to schedule the `map` operator, suppose we have a process like `sumProd` from §4.2 that also performs a `map fn` on the elements before taking the sum and product, for some arbitrary `fn`. Here is the full process description:

```
process sumProdFn (k : &) (s1 : Series k Int)
 with { s2  <- map  k Int fn s1
      ; xs  <= fold k Int s2 with (+) and 0
      ; xp  <= fold k Int s2 with (*) and 1 }
 yields (xs + xp)
```

Using just the first rule in Figure 7 and the one for `map` would produce the following procedure, where we still need to schedule the two `fold` operators:

```
procedure sumProdFn (k : &) (s1 : Series k Int)
 with loop k
      body  { s1_elem   = next k s1
            ; s2_elem   = fn s1_elem }
 yields (xs + xp)                    ** NOT FINISHED
```

Importantly, note that the intermediate variables `s1_elem` and `s2_elem` are named after the corresponding series `s1` and `s2`. In Figure 7 this naming convention is indicated by the superscript on variable names, so $s^{elem}$ is a concrete variable name related to the name $s$. We use this trick to avoid maintaining an environment that maps series names ($s$) to the variable names that bind their individual elements ($s_{elem}$). We similarly relate the names of non-series variables ($x$) to their corresponding accumulators ($x^{acc}$).

Returning to Figure 7, the rule for `fold` adds statements to the nest that first initialize an accumulator, update it in the body of the loop, and then read back the final value. The accumulator lives for the entirety of the loop, so we insert the statements to initialize and read its value in the outer most context, indicated by $\top$.

Scheduling the two fold operations of `sumProdFn` produces the following.

```
procedure sumProdFn (k : &) (s1 : Series k Int)
with loop k
     start { xs_acc    = newAcc 0              *NEW
           ; xp_acc    = newAcc 1 }            *NEW
     body  { s1_elem   = next k s1
           ; s2_elem   = fn s1_elem
           ; xs_acc   := (+) xs_acc s2_elem    *NEW
           ; xp_acc   := (*) xp_acc s2_elem }  *NEW
     end   { xs         = read xs_acc          *NEW
           ; xp         = read xp_acc }        *NEW
yields (xs + xp)
```

When we convert this back to concrete Core code in the next section, we will generate the outer structure of the loop that causes the statements in the body to be evaluated the correct number of times This is governed by the associated rate variable `k`.

#### 4.2.2 Scheduling Pack and Create

Operators from `mkSel` contexts in the process description are scheduled into the body of an abstract `if` statement represented by the `guard` construct of Figure 6. A `guard` binds the rate variable $k_{inner}$ for the inner context, and is also tagged with the rate $k_{outer}$ of the outer context. With `guard`, the `body` and `inner` fields contain more statements to run when the corresponding flag $x_{flag}$ is true. The `inner` nest is needed when a packed series is packed again using a different selector, as this creates a more deeply nested parallel context. The NestedFilter example described in §6 does this.

Figure 8 shows the procedure generated for the explicitly typed version of `filterMax` back in Figure 2, whose process description is in §4.1. On the right of each statement we show the series operator that statement is associated with.

Scheduling the use of `create` has added three separate statements to the nest: one to allocate the new vector buffer; one to write

```
procedure filterMax_d (k1 : &) (s1 : Series k1 Int)
with loop k1
 start { vec'     = newVec k2              *create
       ; mx_acc   = newAcc 0       }      *fold max
 body  { s1_elem  = next k1 s1
       ; s2_elem  = (+ 1) s1_elem         *map (+ 1)
       ; s3_elem  = (> 0) s2_elem  }      *map (> 0)
 inner guard (k2 : &) with k1 s3_elem
       body { s4_elem = s2_elem           *pack
            ; writeVec k2 vec' s4_elem    *create
            ; mx_acc := max mx_acc s4_elem} *fold max
 end   { sliceVec k2 vec'                 *create
       ; mx       = read mx_acc }         *fold max
yields (vec', mx)
```

**Figure 8.** Procedure for `filterMax`

elements into it, and one to destructively slice it down to the final size by overwriting its length field. Note that the use of the $\triangleright$ operator in the corresponding rule from Figure 7 ensures that the result vector is only written to inside the body of the `guard`. Also, note that `newVec` and `sliceVec` statements are tagged with `k2`, which is not in scope in the outer context. This reflects that fact that the length of the result vector cannot be known until the loop completes, as only then will we know how many times the body of the `guard` was entered. We will fix this in the *concretization* phase described in the next section, by rewriting the code to keep track of the length of the result vector.

Interestingly, the `pack` operation becomes a trivial renaming, as shown in the first statement of the inner `body` field of Figure 8. Recall that in the version of `filterMax` from Figure 2, the pack operation was written `s4 = pack k1 k2 Int sel s2`, and here we have scheduled it as just `s4_elem = s2_elem`. It is the `mkSel` function that actually creates the selector context, and that context is expressed here as a `guard` construct. When looking at the source code of `filterMax`, we might imagine that pack would perform some sort of indexing operation, after the discussion in §3.4. However, when the code is expressed in the form of Figure 8 the current index into each series is implicit.

From a logical / type-theory perspective we view `pack` as a *coercion* from a series of the outer rate (`k1`) to a series of the inner rate (`k2`). This coercion is justified by *evidence* that these rates are related, which is expressed as our selector `sel`. However, our compilation process eliminates the selector completely, so the physical packing operation as described in §3.4 is never performed.

### 4.3 Concretization

The concretization phase rewrites constructs that use type level rate variables into ones that use real indices and loop counters.

```
concretize :: [Procedure] -> [ProcedureI]
```

The `ProcedureI` language is very similar to `Procedure` in Figure 6 except that every appearance of a rate variable in `Procedure` is replaced by loop counter or known length in `ProcedureI`.[1] The concrete version of `filterMax` is shown in Figure 9, which is the transformed version of Figure 8.

Deciding whether to change a rate variable to a loop counter (like `k1` to `k1_ix`), accumulator (`k2` to `k2_acc`) or a known length (`k1` to `length s1`) is based on the form of the construct being rewritten. For example, for `sliceVec` we always rewrite its rate variable to a similarly named accumulator.

[1] In our real implementation we use the same data type to represent both, and simply fill-in accumulation variables during concretization.

```
procedure filterMax_c (k1 : &) (s1 : Series k1 Int)
with loopI (k1_ix : Int) (length s1)
 start { vec'     = newVecI (length s1)    *CHANGED
       ; acc      = newAcc  0
       ; k2_acc   = newAcc  0 }            *NEW
 body  { s1_elem  = nextI k1_ix s1         *CHANGED
       ; s2_elem  = (+ 1) s1_elem
       ; s3_elem  = (> 0) s2_elem  }
 inner guardI (k2_ix : Int) k2_acc with s3_elem
                                           *CHANGED
       body { s4_elem = s2_elem
            ; writeVecI k2_ix vec' s4_elem *CHANGED
            ; acc    := max acc s4_elem }
 end   { sliceVecI k2_acc vec'             *CHANGED
       ; mx       = read acc }
yields (vec', mx)
```

**Figure 9.** Concrete Procedure for `filterMax`

The operator `guard (k2 : &) with k1 s3_elem` in Figure 8 changes to `guardI (k2_ix: Int) k2_acc with s3_elem` in Figure 9. For each `guard` we also insert an accumulator (`k2_acc`) to keep track of how many times the guard is entered. The new `guardI` construct executes by first checking the flag `s3_elem`, and if it is true, reads `k2_acc` to get the current entry counter and binds it to `k2_ix` before evaluating the body. In the application of `writeVecI` this index `k2_ix` is then used when constructing the filtered result vector `vec'`. The final `sliceVec` statement is also rewritten to use `k2_acc`, so it knows the final length.

The last task is to rewrite rate variables on `loop` constructs and `newVec` statements to use the lengths of known series. For this we simply look in the environment for a series whose type contains the same rate variable and use its length. If there are multiple series with the same rate variable then we just choose the first one — as all series tagged with the same rate are guaranteed to have the same length.

### 4.4 Extracting Implementation Code

The Extract phase takes our concretized list of procedures and converts them back to a module of imperative flavoured Core code. We end up with a top-level binding for each `Procedure`, which mirrors the `slurp` phase from §4.1.

```
extract     :: [Procedure] -> Module
```

The extracted code for `filterMax` is shown in Figure 10. As we can see, this final phase is again mostly a change of syntax. The real work of fusion has been performed by the scheduling phase, and the concretization pass in the previous section has already reduced the abstraction level of our program to something that looks like real loop code.

In the extracted code the `loopI` and `guardI` constructs have changed to calls to similarly named functions. As we will see in the next section, these can be implemented as Haskell library functions and then inlined, or transformed to into tail recursive loops as discussed in §5.2.

## 5. Details of the Conversion

The previous section contains the main details of the fusion transformation, starting with a high level description of the computation to be performed, and ending in imperative loop code.

As GHC Core is a pure functional language, the imperative code in Figure 10 is not the end of the story. In our implementation we express the code in Figure 10 in an imperative version of GHC Core named Core Flow. This language is essentially the same as

```
filterMax_x :: ∀(k1 : &)
            .  Series k1 Int -> (Vector Int, Int)
 = Λ(k1 : &).
   λ(s1 : Series k1 Int).
   let vec'   : Vector Int = newVec @Int (length s1)
   let acc    : Ref Int    = newRef @Int 0
   let k2_acc : Ref Int    = newRef @Int 0
   let _ : Unit
       = loopI (length s1)
         (λ(k1_ix : Int).
          let s1_elem = next @k1 @Int s1 k1_ix
          let s2_elem = add  @Int 1    s1_elem
          let s3_elem = gt   @Int 0    s2_elem
          let _  : Unit
              = guardI k2_acc s3_elem
                 (λ(k2_ix : Int).
                  let s4_elem = s2_elem
                  let _ = writeVec @Int
                            vec k2_ix s4_elem
                  let _ = writeRef @Int acc
                            (max (readRef @Int acc)
                                 s4_elem)
                  in ())
          in ())
   let k2_ix : Int        = readRef  @Int k2_acc
   let vec'' : Vector Int = sliceVec @Int k2_ix vec'
   let mx    : Int        = readRef  @Int acc
   in  (vec'', mx)
```

**Figure 10.** Extracted Imperative Core Code for `filterMax`

GHC Core, being a version of System-F, except that it is strict, has untracked side effects and includes imperative functions like `newVec` and `readRef` as primitive operators.

### 5.1 State Threading

As GHC uses monadic state threading to sequence effectful statements, we must thread GHC's primitive world token though the extracted code before converting it back to real GHC Core. We have implemented this state threading transform generically, so it is parameterized by two sets of type signatures: one that assumes a language with untracked side effects, and one that uses state threading. For example, the two versions of the signature for `writeVec#` are as follows, where `writeVecE#` has untracked effects and `writeVecW#` uses a world token of type `W`.

```
writeVecE# :: ∀(a:*). Vector a -> Int -> a -> ()
writeVecW# :: ∀(a:*). Vector a -> Int -> a -> W -> W
```

### 5.2 Loop Winding

After the extracted code of Figure 10 has had the world token threaded through it, it can converted back to real GHC Core and type checked.

In our implementation we originally wrote `newVec`, `newRef`, `loopI`, `guardI`, `next` and so on as standard Haskell library functions. Although this allows the program to run, the fact that GHC does not track pointer aliasing between heap objects results in inefficient object code when using mutable references.

To avoid this problem, we instead perform a *loop winding* transformation on the code that converts uses of `loopI` and `guardI` into real tail recursive loops, and mutable references into accumulating parameters. This transform is ad-hoc because it assumes that mutable references do not escape the extracted function, and that there is no additional aliasing between reference variables like `acc` and `k2_acc`. However, because we generated the code ourselves we know these properties are true.

### 5.3 Primitive Arithmetic and Unboxed Types

Unlike GHC Core, the Core Flow language does not make a distinction between boxed and unboxed types [17]. When we `slurp` a `Process` description from the original GHC Core program we require that program to use boxed numeric values and operators. However, when we convert extracted code *back* to GHC Core, we use the unboxed versions. Unboxed primitive operators typically compile down to single machine instructions. To handle the impedance mismatch we then generate a wrapper function that marshals between the signature of the original source function and the extracted version. For example, the wrapper for `filterMax` function would be:

```
filterMax = Λ(k : &). λ(s : Series k Int).
            case filterMax_x s of
             (# vec, n #) -> (vec, I# n)
```

Standard unboxing techniques guided by strictness information *usually* work, but as strictness analysis is conservative the unboxing is not guaranteed. When the rest of the loop body has been fused well enough to execute in only a handful of cycles, the cost of a single unboxing operation in an inner loop can easily dominate program runtime. Brutally converting arithmetic operations from their boxed to unboxed versions during flow fusion ensures that we never pay the price of thunk entry in fused code.

## 6. Benchmarks

Benchmarks were conducted on a MacBook Pro with 2.8GHz Intel Core i7 with 8GB of RAM. Source code is available from the `repa-plugin` darcs repository.

We use micro-benchmarks because our fusion system addresses these specific programming patterns, rather than being an improvement on the ambient performance of the program — as with optimizations like pointer tagging [15].

For each benchmark we compare four implementations:

- *Stream*: using stream fusion [7] and unboxed vectors;[2]
- *Flow*: using our new Flow fusion framework;
- *Unfused Flow*: using our Flow API but without the plugin;
- *Hand-fused*: hand written and fused C code.

The Unfused Flow versions use the exact same code as the Flow versions, except they are compiled without the plugin that actually performs the fusion transformation. In this case the benchmarks are compiled via a fallback implementation of the user-facing API of Figure 3, implemented in terms of standard stream fusion [7]. The fallback implementation provides a quick compilation path for people that do not want to install the plugin or care about the last ounce of performance, as well as a convenient way of testing the plugin itself.

### 6.1 Dot Product

A pair of two-dimensional vectors are multiplied element-wise and the results summed. Each two dimensional vector is stored as two arrays of integers, giving four arrays in total. As discussed in §2.2, code compiled with stream fusion produces a loop counter for each vector. With flow fusion the concretization phase (§4.3) naturally causes loop counters to be shared. This provides a 25% speedup over stream fusion and puts us on par with the reference C implementation.

[2] from the `vector` library on Hackage

| Benchmark | Input size | Stream (ms) | Flow (ms) | | Unfused Flow (ms) | | Hand-fused C (ms) | |
|---|---|---|---|---|---|---|---|---|
| Dot Product | $10^8$ | 655 | 489 | (75%) | 1,096 | (167%) | 474 | (72%) |
| MapMap | $10^8$ | 842 | 636 | (75%) | 842 | (100%) | 615 | (73%) |
| FilterSum | $10^8$ | 505 | 430 | (85%) | 1,132 | (224%) | 344 | (68%) |
| FilterMax | $10^8$ | 567 | 521 | (91%) | 1,496 | (263%) | 360 | (63%) |
| NestedFilter | $10^8$ | 485 | 420 | (86%) | 1,202 | (247%) | 376 | (77%) |
| QuickHull | $10^7$ | 419 | 208 | (49%) | 857 | (204%) | 183 | (43%) |

**Figure 11.** Benchmark Results for Flow Fusion

### 6.2 MapMap

The elements of a vector of integers are doubled, and the resulting vector has a constant added in one operation and subtracted in another. With stream fusion the first result is materialized in memory, and then read back by each of the subsequent vector operations. With flow fusion the first result is not materialized, which also puts us on par with the reference C implementation.

### 6.3 FilterSum

A vector of integers is filtered and the elements of the original and result vectors are separately summed. The result vector and both sums are returned in a 3-tuple. With stream fusion the code uses three separate loops, one for each operator. With flow fusion the code uses a single loop that sums both the original and result vectors while constructing the result.

Although the Core code produced by flow fusion is optimal, the low-level object code suffers from pointer aliasing problems. The back-end code generator does not know that the input vector does not alias with the result vector, nor that writes to the element data of the result do not affect its starting offset field. Ultimately, the length field of the first vector, and starting offsets for both vectors are repeatedly read in the inner loop. The C compiler can infer correct aliasing information, and thus saves three memory reads per loop iteration.

### 6.4 FilterMax

This is the `filterMax` example described earlier. With stream fusion the filtered result vector must be read back from memory to sum its elements. With flow fusion the sum is performed in the same loop as the filter. Similarly to the FilterSum benchmark, although the Core code is optimal, low level pointer aliasing problems in the object code prevent us from matching the performance of the C version.

### 6.5 NestedFilter

An input vector is filtered, this result filtered again by another predicate, and both results are returned in a tuple. With stream fusion the first result is constructed in memory and then read back to perform the second filter. With flow fusion both results are constructed in the same loop and the first is not read back. As mentioned in §4.2.2 this benchmark uses two nested applications of `mkSel`, thus the Core code contains two nested guards.

### 6.6 QuickHull

QuickHull finds the smallest convex hull of a set of points in the 2-d plane. The algorithm operates in two phases. The first is an initialization phase where we determine the left-most and right-most points in the input set. If these results are computed in two separate fold operations then stream fusion cannot fuse this code. In the second phase, we need to determine the set of points above a cutting line and also the point furthest from it. This is a `filterMax`-like operation that stream fusion can also not fuse.

## 7. Related Work

Our work sits between the fields of array fusion and compilation of data flow languages. As mentioned in §3.5 the Flow Fusion API from Figure 3 defines a domain specific, first order, non-recursive, synchronous, finite, data flow language for writing array programs. This language is a fragment of a more general data flow language such as Lustre [1] or Lucid Synchrone [19], and our rate types are similar to the clock types of Lucid, and Kahn networks [3].

The full data flow languages are intended primarily for implementing embedded control systems, writing signal processing circuits as code. In this context, support for infinite streams is essential, as the input signal must be processed indefinitely. Clock typing systems for these languages ensure *causality* and *syncronicity* in the presence of recursive data flows and infinite streams. Causality ensures the system has a well defined notion of time, and syncronicity ensures that it can be evaluated without needing unbounded buffers of intermediate signal data. In contrast, as we deal only with acyclic graphs our programs are automatically causal, and the finiteness of arrays eliminates the need for unbounded buffering.

The idea of representing a loop as an *anatomy* of `start`, `body`, `inner` and `end` fields as in our `Procedure` language of Figure 6 appears in Shivers's work [22] as well as Waters's work on series expressions [27]. Shivers's work focuses on taming the vagarities of variable scoping in Common LISP loop generation macros [23]. He gives an eight-field loop anatomy and scoping rules for the source language in terms of this anatomy. The source code using these macro packages is more loopish in nature than our functional code using `map`, `fold`, `pack` and so on. In the former, one writes `loop` to introduce a new one, and then adds modifiers to specify what results should be computed.

The work most closely related to ours is Waters's series expression framework [27]. In its essence, the paper you are reading straps a cut down clock calculus to a functionally flavoured version of Waters's compilation method, and bakes it into a GHC plugin. Waters's work does not use explicit clock or rate typing information, and his compilation method generates Pascal code with `goto` statements and fresh labels. Full featured clock typing systems like [2] (1995) and [3] (1996) did not appear until after Waters completed his line of work on series expressions [27] (1991).

In place of a rate or clock typing system, Waters's framework uses *the online criteria*, which is one of four restrictions he gives that govern whether an *optimizable series expression* can be fused. The others are 1) series expressions are not subjected to any conditional or looping control flow (discussed in §4.1.2); 2) the program is statically analysable (meaning compound operators are inlined or otherwise visible to the compiler (§3.6), and 3) series are not consumed in a random access manner (§3.1).

The online criteria says that every non directed cycle in the data flow graph must consume and produce its elements in lockstep. Using rate typing we rephrase this by saying every series in the cycle must have the same rate. As we have a non-recursive `Process` language, the only way to make a cycle in the graph would be to use

an operator with multiple inputs, but the only one is `map2` whose type requires its input and output series to have the same rate.

Finally, the fact that short cut fusion cannot deal with branching data flows stems from a deeper problem: no sequential evaluator can perform a lazy `unzip`-like operation in a space efficient way. Given `unzip x = (map fst x, map snd x)`, while computing the elements of the first component of the result, we cannot reclaim the space for the input `x` elements because we will need those again to compute the second component of the result. Ideally, we would alternate between the two components, pulling one element from each side after the other, but lazy evaluation does not do this.

Hughes gives an informal proof of the above fact in [12] (1983), where a "sequential evaluator" is defined as one that, once the evaluator has begun to reduce an expression $E$, it will only reduce $E$ and other expressions that $E$ demands until $E$ has been completely reduced. From this fact we infer that no system based on partial evaluation of purely functional code can fuse branching data flows when the underling language has a sequential semantics (as does GHC Core). Our flow fusion system steps around this problem by splitting the functional array operators into imperative code, and then interleaving the various components. We perform the job of a concurrent scheduler at compile time.

## 8. Future Work

This paper only discusses a few array combinators: `map`, `fold`, `pack` and so on, but others such as `append` and `scan` can be implemented in a similar way. For example, `append` is an instance of the more general `combine` combinator, that takes a series of flags, two series of elements, and then chooses which element to return based on the flag:

```
combine [T F T T F] [1 2] [3 4 5] = [3 1 4 5 2]
```

After §3.4, the vector of flags would be implemented as an extended selector `Sel2 k1 k2 k3` that relates three separate rates: the rate of the flags, the rates of the two input vectors of elements. In the `Procedure` language, this new selector context would be compiled as a real `if` construct with both a `then` and `else` branches, unlike our `guard` construct that only has the `then` branch.

In future work we will perform further transformations at the `Procedure` level to introduce SIMD instructions and multicore evaluation. Unlike the loop parallelization systems in imperative language compilers, we do not need an iteration dependency analysis. Our our loops lack such dependencies by construction.

***Acknowledgements*** Thanks to Barry Jay for the suggestion to phrase the problem with short cut fusion in terms of what information is available to the program transformations; Peter Gammie for the connection with John Hughes's early work; Simon Peyton Jones and Geoff Mainland for comments on a draft version of this paper. This work was supported in part by the Australian Research Council under grant number LP0989507.

## References

[1] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL: Principles of Programming Languages*. ACM, 1987.

[2] Paul Caspi and Marc Pouzet. A Functional Extension to Lustre. In *International Symposium on Languages for Intentional Programming*. World Scientific, 1995.

[3] Paul Caspi and Marc Pouzet. Synchronous Kahn networks. In *ICFP: International Conference on Functional Programming*. ACM, 1996.

[4] Manuel M. T. Chakravarty and Gabriele Keller. Functional array fusion. In *ICFP: International Conference on Functional Programming*. ACM, 2001.

[5] Siddhartha Chatterjee, Guy E. Blelloch, and Allan L. Fisher. Size and access inference for data-parallel programs. In *PLDI: Programming Language Design and Implementation*. ACM, 1991.

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

[7] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. In *ICFP: Internal Conference of Functional Programming*. ACM, 2007.

[8] Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. In *FPCA: Functional Programming Languages and Computer Architecture*. ACM, 1993.

[9] Clemens Grelck, Karsten Hinckfuß, and Sven-Bodo Scholz. With-loop fusion for data locality and parallelism. In *IFL: Implementation and Application of Functional Languages*. Springer-Verlag, 2006.

[10] Nicolas Halbwachs, Pascal Raymond, and Christophe Ratel. Generating efficient code from data-flow programs. In *PLILP: Programming Langauge Implementation and Logic Programming*, 1991.

[11] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data traversals. In *ICFP: International Conference on Functional Programming*. ACM, 1997.

[12] John Hughes. *The Design and Implementation of Programming Languages*. PhD thesis, Programming Research Group, Oxford University, July 1983.

[13] Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, and Ben Lippmeier. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *ICFP: International Conference on Functional Programming*. ACM, 2010.

[14] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *PLDI: Programming Language Design and Implementation*. ACM, 1994.

[15] Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones. Faster laziness using dynamic pointer tagging. In *ICFP: International Conference on Functional Programming*, 2007.

[16] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA: Functional Programming Languages and Computer Architecture*. ACM, 1991.

[17] Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *FPCA: Functional Programming and Computer Architecture*. ACM, 1991.

[18] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*. Schloss Dagstuhl, 2008.

[19] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006.

[20] Tiark Rompf et al. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *POPL: Principles of Programming Languages*. ACM, 2013.

[21] Vivek Sarkar and Guang R Gao. Optimization of array accesses by collective loop transformations. In *International Conference on Supercomputing*. ACM, 1991.

[22] Olin Shivers. The anatomy of a loop. In *ICFP: International Conference on Functional Programming*. ACM, 2005.

[23] Guy Steele. *Common Lisp the Language*. Digital Press, 1990.

[24] Philip Wadler. Listlessness is better than laziness. In *LISP and Functional Programming*, 1984.

[25] Joe D. Warren. A hierarchical basis for reordering transformations. In *POPL: Principles of Programming Languages*. ACM, 1984.

[26] Richard C. Waters. Efficient interpretation of synchronizable series expressions. In *PLDI: Programming Language Design and Implementation*. ACM, 1987.

[27] Richard C. Waters. Automatic transformation of series expressions into loops. *TOPLAS: Transactions on Programming Languages and Systems*, 13(1), 1991.