
# Faster Coroutine Pipelines


MICHAEL SPIVEY, University of Oxford

Coroutine pipelines provide an attractive structuring mechanism for complex programs that process streams of data, with the advantage over lazy streams that both ends of a pipeline may interact with the I/O system, as may processes in the middle. Two popular Haskell libraries, *Pipes* and *Conduit*, support such pipelines. In both libraries, pipelines are implemented in a direct style by combining a free monad of communication events with an interpreter for (pseudo-)parallel composition that interleaves the events of its argument processes. These implementations both suffer from a slow-down when processes are deeply nested in sequence or in parallel. We propose an alternative implementation of pipelines based on continuations that does not suffer from this slow-down. What is more, the implementation is significantly faster on small, communication-intensive examples even where they do not suffer from the slow-down, and comparable in speed with similar programs based on lazy streams. We also show that the continuation-based implementation may be derived from the direct-style implementation by algebraic reasoning.

CCS Concepts: • **Software and its engineering → Functional languages**; **Coroutines**;

**ACM Reference format:**
Michael Spivey. 2017. Faster Coroutine Pipelines. *Proc. ACM Program. Lang.* 1, 1, Article 5 (September 2017), 23 pages.
https://doi.org/10.1145/3110249

## 1 INTRODUCTION

Traditionally, functional programmers have built processing pipelines by composing functions that map streams to streams, and representing the streams as lazy lists. This style provides a powerful way of structuring complex programs, and it is supported by Haskell's I/O system, which provides a function *readFile* that yields the contents of an entire file as a lazy list of characters. For example, we can program a lexer that takes a stream of characters and produces a stream of tokens, a parser that takes a stream of tokens and produces a stream of expressions, and an evaluator that takes an expression and evaluates it, and put together a simple language implementation as



$$system = concat \cdot map\, evaluate \cdot parser \cdot lexer$$

We can use this in a main program such as

$$main = \mathbf{do}\ text \leftarrow readFile\,\text{"prog.txt"};\ putStr\,(system\ text)$$

Note, however, that *readFile* depends crucially on laziness to be usable for large files. Even in Haskell it cannot be defined in terms of simpler I/O primitives such as *readChar*, and this makes it inflexible if we want a slightly different behaviour. For example, we might like to make an interactive program that prints a prompt for each line of input, and there is no way to do this and still have the text available as a lazy stream. Worse still, there is no way for the parser to print output of its own, such as to report syntax errors, without the cooperation of processes further down the pipeline.


2475-1421/2017/9-ART5 $15.00
https://doi.org/10.1145/3110249

We can see why Haskell's monadic I/O primitives cannot be used to implement *readFile* by considering a program that, say, inputs a file and uses *filter* to print all the characters except digits. With *readFile*, input and output actions will be *implicitly* interleaved as the program demands input data, with the demand begin controlled by a function *filter* that operates on lists of characters and knows nothing about the I/O process that produces the list. The monadic I/O primitives, on the other hand, enforce an *explicit* ordering of I/O actions, as can be seen by considering the existence of an implementation where the I/O state is threaded through the program. If a program inputs from one file and outputs to another, then the interleaving of the input and output actions must be determined by the program, and this cannot be done by considering the two files separately.

These limitations of stream-based I/O have led programmers to want more general frameworks for stream processing, where each component in a pipeline can communicate explicitly on channels to its left and right, but also perform out-of-band I/O actions of its own. It then becomes possible for both ends of a pipeline to interact with the I/O system, perhaps reading an input file and writing an output file in explicit chunks, or prompting for input as it is required by the program. Both the *Pipes* [Gonzalez 2012] and *Conduit* [Snoyman 2011] packages for Haskell provide this kind of facility.

In this paper, we give two implementations of a pipeline-like toolkit that is a bit simpler than the ones presented by these packages, providing a monad for composing sequential processes, and also an operation ‖ that takes two processes and joins the output channel of one to the input channel of the other, making them behave like a single process. Pipelines are attractive as a presentation of coroutines because they have a natural scheduling strategy – demand-driven evaluation – that avoids any need to introduce operating-system-like data structures and arbitrary scheduling algorithms in the implementation. Working with them feels more like exploring the semantics of a language than implementing an operating system.

The first implementation we give is similar to the existing Haskell packages, and uses an interpreter or *trampoline* to execute a pipeline step-by-step, with (broadly speaking) one bounce of the trampoline for each communication step done by the pipeline. But trampolining is known as a technique for getting the effect of tail recursion in languages that don't provide it directly, and specifically as a way of enabling us to program in continuation-passing style in such languages; and Haskell is not a language that is defective in its support for tail recursion. So why not look for an implementation of pipelines that uses continuations directly? That is the idea behind the second implementation given here. It represents the context of a process using a pair of continuations, either of which can be passed as an argument to the other; this presents a tricky but solvable problem with types.

After showing the two implementations, we continue with an analysis of their efficiency, together with experimental results that show that the continuation-based implementation is consistently faster on small test programs, and in some circumstances offers a linear order of growth of running time where the direct-style program suffers quadratic growth. The next section discusses the relationship between the implementations as revealed by a representation function that converts from one to the other, allowing the better implementation to be derived systematically. Then we show how the continuation-based toolkit can be extended beyond strictly linear pipelines to allow forks and joins.

Both the *Pipes* and *Conduit* libraries are equipped with rewrite rules that are intended to allow the Haskell compiler to transform away the intermediate streams in simple cases and produce a program in iterative form. We do not consider these rules here, but since they depend on algebraic properties of the combinators and not on their implementation or the representation of processes, they should transfer to both of the implementations considered here.

This paper uses the notation of Haskell, and it is assumed that the reader has a basic familiarity with Haskell and with the use of monads to structure programs that exploit computational effects. However, the designs we discuss could be replicated in other functional languages that support tail calls, and (except where explicitly noted) no reliance is made on the laziness of Haskell.

## 2 PIPELINES

Before presenting the two implementations of a pipeline toolkit, we will define the interface that both of them will satisfy, and give some examples of programs that could be run with either of them.

We shall study a class of type constructors $pipe\,\iota\,o\,\alpha$ with the following properties:

- A value of type $pipe\,\iota\,o\,\alpha$ represents a process with a single *input channel* capable of transmitting values of type $\iota$, a single *output channel* for values of type $o$, and the possibility of *terminating* and yielding a value of type $\alpha$.
- For any $\iota$ and $o$, the type constructor $pipe\,\iota\,o$ is a monad, with an operation $return :: \alpha \to pipe\,\iota\,o\,\alpha$ that immediately terminates, yielding the value passed as an argument, and an operation

  $$(\gg\!=) :: pipe\,\iota\,o\,\alpha \to (\alpha \to pipe\,\iota\,o\,\beta) \to pipe\,\iota\,o\,\beta$$

  that composes two processes sequentially, running the first one until it terminates, then passing the value it yields to the second one.
- There are processes $input :: pipe\,\iota\,o\,\iota$ and $output :: o \to pipe\,\iota\,o\,()$ that communicate on the input and output channels. Both terminate as soon as the communication has happened, with *output* taking as a parameter the value it should send, and *input* yielding as its result the value it has received.
- Two processes $p :: pipe\,\iota\,\eta\,()$ and $q :: pipe\,\eta\,o\,()$ can be combined into a single process $p \parallel q :: pipe\,\iota\,o\,()$, by joining the output channel of $p$ to the input channel of $q$ and hiding it, so that the combined process has an input channel internally connected to $p$ and an output channel internally connected to $q$. The symbol $\parallel$ is intended to suggest the pipe operator in the UNIX shell. The result type of $()$ for the processes $p$ and $q$ encode the fact that $p$ and $q$ cannot produce any sensible return value, and in our implementations it becomes an error if they ever do terminate.
- If $e$ is an I/O action of type $IO\,\alpha$, then $effect\,e :: pipe\,\iota\,o\,\alpha$ is a process that performs the action $e$ and returns the resulting value. Also, $exit :: pipe\,\iota\,o\,\alpha$ is a process that causes the entire processing pipeline to terminate.

The declarations of *input*, *output*, $\parallel$, *effect* and *exit* are packaged in the following class declaration:

```
class ∀ ι o · Monad (pipe ι o) ⇒ PipeKit pipe where
  input :: pipe ι o ι
  output :: o → pipe ι o ()
  (||) :: pipe ι η () → pipe η o () → pipe ι o α
  effect :: IO α → pipe ι o α
  exit :: pipe ι o α
```

If *pipe* is an instance of *PipeKit*, then we expect also that $pipe\,\iota\,o$ will be a monad for each choice of types $\iota$ and $o$: that is the intended implication of the universally quantified constraint shown above. Although such constraints are not part of the Haskell type system, there is a way of expressing them with the following coding trick, due to Trifonov [2003]. First, we introduce a type class *Monad_f*, like *Monad* but with extra type parameters.

```
class Monad_f m where
  return_f :: α → m ι o α
  bind_f :: m ι o α → (α → m ι o β) → m ι o β
```

If $m$ is an instance of *Monad_f* then naturally $m\,\iota\,o$ is an instance of *Monad*.

```
instance Monad_f m ⇒ Monad (m ι o) where
  return = return_f
  (>>=) = bind_f
```

With these definitions in place, we can replace the quantified constraint $\forall\,\iota\,o \cdot \mathit{Monad}\,(\mathit{pipe}\,\iota\,o)$ with the simple constraint *Monad_f pipe*, and replace all instance declarations for $\mathit{Monad}\,(\mathit{pipe}\,\iota\,o)$ that appear below with the corresponding instance declarations for *Monad_f pipe*. This trick depends on the *FlexibleInstances* extension of GHC.

Simple examples of processes include *upfrom n*, which outputs the integers $n$, $n+1$, …, and *filter test*, which forever inputs values and passes on those that satisfy a boolean test. As with most processes we shall define, these are generic in the pipeline toolkit.

```
upfrom :: PipeKit pipe ⇒ Int → pipe () Int ()
upfrom n = do output n; upfrom (n + 1)

filter :: PipeKit pipe ⇒ (α → Bool) → pipe α α ()
filter test = forever (do x ← input; if test x then output x else skip)
```

The helpers *forever* and *skip* are defined as follows.

```
forever :: Monad m ⇒ m α → m β
forever p = q where q = p >> q

skip :: Monad m ⇒ m ()
skip = return ()
```

We can form the composite process *upfrom* 0 || *filter odd* to make a process that outputs only the odd integers. Note that neither this process nor its components ever terminate, and this is normal for processes that are composed in a pipeline. An individual action like *input* does terminate, returning the value that has been received, and we exploit this in defining *filter*, combining *input* and *output* with the **do** notation to describe the cyclic action that *filter* performs.

For simplicity of implementation, processes that are directly combined into a pipeline using || are actually forbidden from terminating, and we will rely on *exit* to halt the program instead. If desired, it would be simple enough to adopt the convention that a process that terminates sends a sentinel value *bye* to the process to its right, just by replacing each process $p$ with the process $p \gg \mathit{forever}\,(\mathit{output}\,\mathit{bye})$. We could additionally adopt the convention that all the values transmitted on channels have type $\mathit{Maybe}\,\alpha$ and set $\mathit{bye} = \mathit{None}$, but these are options independent of the other choices we face. Alternatively, the distinction between processes that may terminate and those that must not do so can be made more rigid by introducing an empty type *Empty* and replacing the type $\mathit{pipe}\,\iota\,o\,()$ that we have used for non-terminating processes with $\mathit{pipe}\,\iota\,o\,\mathit{Empty}$, using the type system to make it impossible to use *return* in defining such a process.

We can use these operations to write interesting programs. For example, here is the definition of a process *primes n* that outputs the first $n$ primes by repeatedly filtering the stream of natural numbers.

```
primes :: PipeKit pipe ⇒ Int → pipe () Int ()
primes n = upfrom 2 || sieve || Pipe.take n
```

```
sieve :: PipeKit pipe ⇒ pipe Int Int ()
sieve = do p ← input; output p; Pipe.filter (λ x → x mod p ≠ 0) || sieve
```

This uses the utility function *filter* defined earlier, which we will begin writing as *Pipe.filter* to avoid confusion with the well-known function *List.filter* acting on lists. Another function *Pipe.take n* exits (stopping the whole program) after passing on *n* values.

```
take :: PipeKit pipe ⇒ Int → pipe α α ()
take n = if n ≡ 0 then exit else (do x ← input; output x; take (n − 1))
```

This program at least (since it performs no input) could have been written as a conventional composition of functions on lazy streams, but it illustrates well enough the feeling of the programming style. We will compare it for speed with the conventional version later.

It would take us too far afield to present an entire interactive language implementation, though that was the author's motivation for studying pipelines. As a substitute, here is a program that prompts for lines of input with "<", reverses each one, and prints the reversed line labelled with ">". Note that both *readLine* and *putLine* use *effect* to interact with the I/O system.

```
rev :: PipeKit pipe ⇒ pipe () () ()
rev = forever readLine || Pipe.map reverse || forever putLine

readLine :: PipeKit pipe ⇒ pipe u String ()
readLine = do s ← effect (do putStr "< "; getLine); output s

putLine :: PipeKit pipe ⇒ pipe String v ()
putLine = do s ← input; effect (putStrLn ("> " ++ s))
```

This uses another generic utility function *Pipe.map*, defined as follows.

```
map :: PipeKit pipe ⇒ (a → b) → pipe a b ()
map f = forever (do x ← input; output (f x))
```

With these examples as motivation, we now turn to the task of implementing the pipeline toolkit.

## 3 A DIRECT IMPLEMENTATION

The first implementation represents each process as a function that returns its first action as a concrete command, with the future behaviour of the process embedded within it as another process value. Sequential and parallel composition are then implemented by case analysis on the processes being combined, interpreting their input and output actions in an appropriate way.

In this style of implementation, a process has type *DirectPipe ι o a*, defined as follows:

```
data DirectPipe ι o α =
    Input (ι → DirectPipe ι o α)
  | Output o (DirectPipe ι o α)
  | Done α
```

The constructors *Input* and *Output* represent processes that have paused waiting for input or output.

- In the case *Input h*, the argument *h* is a function to apply to an input value *v* when it arrives.
- In the case *Output v r*, the value *v* is being output, and *r* represents the rest of the process after the output.

In both cases, part of the value is another process *h v* or *r* of type *DirectPipe ι o α*, representing a suspended state. If we care, we can rely on laziness to ensure that this suspended state is not

explored further until the next action is needed, or in a strict language we could write "$\lambda() \rightarrow$" systematically to make the delay explicit.

- As a third option, the value $\mathit{Done}\, x$ represents a process that has terminated and yielded a value $x$.

For a full implementation of the pipeline toolkit, we also need constructors for *DirectPipe* that correspond to *effect* and *exit*.

$$\begin{aligned}&\textbf{data}\ \mathit{DirectPipe}\, \iota\, o\, \alpha = \ldots\\ &\quad \mid\ \forall\, \beta \cdot \mathit{Effect}\, (\mathit{IO}\, \beta)\, (\beta \rightarrow \mathit{DirectPipe}\, \iota\, o\, \alpha)\\ &\quad \mid\ \mathit{Exit}\end{aligned}$$

Note that the *ExistentialQuantification* extension of GHC is needed here. Supporting *effect* and *exit* also requires additional cases in each function that consumes *DirectPipe* values, and specifically in the implementations of $\gg\!=$ and $\|$ that are given below. This code is omitted in the paper because it adds nothing to the discussion, but it is included in the accompanying code archive.

We need to make $\mathit{DirectPipe}\, \iota\, o$ into a monad, and this means considering how to turn $p \gg\!= f$ into a process, where $p$ is itself a process represented as above, and $f$ is a function that can consume any value $x$ yielded by $p$. Both $p$ and $f\, x$ will share the same input and output channels, so that input or output by either of them will become input or output by the whole process. (We will consider below the different operation $\|$ that interleaves two processes and connects their channels together.)

To define $p \gg\!= f$, we can use case analysis on the value $p$. It is helpful to consider first the case $p = \mathit{Output}\, v\, r$; here $v$ is a value output by $p$, and $r$ captures its future behaviour. It's clear that the value $v$ should become an output of the compound process $p \gg\!= f$; but what about the future behaviour of that process? We still have the remainder $r$ of $p$, and after that we still have $f$ to do: so the correct action of the compound process is $\mathit{Output}\, v\, (r \gg\!= f)$.

Similar reasoning applies when $p$ performs an input action $\mathit{Input}\, h$, where the handler function $h$ will receive whatever value is input. In this case, the correct behaviour is to take the input $v$ and form $h\, v \gg\!= f$, so the compound process should be $\mathit{Input}\, (\lambda\, v \rightarrow h\, v \gg\!= f)$. Putting these ideas together gives the following instance declaration.

$$\begin{aligned}&\textbf{instance}\ \mathit{Monad}\, (\mathit{DirectPipe}\, \iota\, o)\ \textbf{where}\\ &\quad \mathit{return}\, x = \mathit{Done}\, x &(1)\\ &\quad p \gg\!= f = &(2)\\ &\qquad \textbf{case}\ p\ \textbf{of}\\ &\qquad\quad \mathit{Input}\, h \rightarrow \mathit{Input}\, (\lambda\, v \rightarrow h\, v \gg\!= f)\\ &\qquad\quad \mathit{Output}\, v\, r \rightarrow \mathit{Output}\, v\, (r \gg\!= f)\\ &\qquad\quad \mathit{Done}\, x \rightarrow f\, x\end{aligned}$$

Note that the process $p \gg\!= f$ becomes $f\, x$ when $p$ terminates; but until that point, every input or output action of $p$ is reflected as an input or output of the compound process, and a composition $h\, v \gg\!= f$ or $r \gg\!= f$ survives to deal with the next input or output action.

Having made $\mathit{DirectPipe}\, \iota\, o$ into a monad, we can now turn attention to the operations specific to *PipeKit*. For *input* and *output*, we can use the corresponding constructors of the *DirectPipe* type fairly directly.

$$\begin{aligned}&\textbf{instance}\ \mathit{PipeKit}\ \mathit{DirectPipe}\ \textbf{where}\\ &\quad \mathit{input} = \mathit{Input}\, (\lambda\, v \rightarrow \mathit{return}\, v) &(3)\\ &\quad \mathit{output}\, v = \mathit{Output}\, v\, (\mathit{return}\, ()) &(4)\end{aligned}$$

To implement the operation $p \,\|\, q$ we will again need to use case analysis on the form of a process. To make the evaluation nicely demand-driven, it seems best to begin with $q$. If $q$ starts by outputting,

*Output v r*, then so does the pipe, and the case is easily dealt with as *Output v* (*p* ‖ *r*). If *q* inputs as *Input h*, on the other hand, we need to activate *p*, keeping the handler function *h* until such time as *p* outputs. Running *p* under these conditions is the function of the helper function ‖′ in the following code.

```
instance PipeKit DirectPipe where ...
  p ‖ q =                                                    (5)
    case q of
      Input h    → p ‖′ h
      Output v r → Output v (p ‖ r)
      Done x     → error "terminated"
    where
      p ‖′ h =                                               (6)
        case p of
          Input g    → Input (λ v → g v ‖′ h)
          Output v r → r ‖ h v
          Done x     → error "terminated"
```

We note again that the definition of *p* ‖ *q* forms a new process with the shape *p* ‖ *r* or *r* ‖′ *h* after each input or output action.

To make these pipelines useful, we need to be able to connect them to the outside world, and this is the point where we meet the top-level trampoline: at each bounce, it performs an input or output action by interpreting it in terms of the *IO* monad, and loops to find the next action.

```
runDirPipe :: (Read ι, Show o) ⇒ DirectPipe ι o ( ) → IO ( )
runDirPipe p =
  case p of
    Input h    → (do v ← readLn; runDirPipe (h v))
    Output v r → (do print v; runDirPipe r)
    Done ( )   → skip
```

Though they are useful for testing, strictly speaking these implicit connections between the ends of a pipeline and the I/O system are superfluous, because we can take any pipeline and sandwich it between processes like *readLine* and *putLine* (defined in Section 2) that make the connections explicit, and can be adapted to read from and write to any file.

In this direct implementation, if we form a lengthy pipeline of processes such as

$$(p_1 \parallel p_2) \parallel (p_3 \parallel (p_4 \parallel p_5)),$$

then an input action of $p_1$ or an output action of $p_5$ must filter to the top level and result in an external input or output action. However, when (say) $p_4$ wants to input, the communication is mediated by the instance of ‖ that is its common ancestor with $p_3$, and the input request must propagate only that far. Communication between $p_2$ and $p_3$ must propagate to the outermost instance of ‖, while that between $p_1$ and $p_2$ uses the ‖ instance that immediately encloses them. Functionally, the ‖ operation is associative, but as we will see later, the efficiency of a program using this implementation of pipes can depend crucially on the way in which the pipeline is put together.

## 4 A CONTINUATION-BASED IMPLEMENTATION

A second implementation of the pipeline toolkit makes heavy use of continuations, in particular to represent the saved state of suspended processes. Each process will become a function that accepts

*three* continuations, one for input, a second for output, and a third that is called when the process terminates.

To set up the framework, we can begin with a monad that turns ordinary monadic programs into continuation-passing style. In this style, the type $\theta \to \phi$ of functions from $\theta$ to $\phi$ is replaced by the type

$$\theta \to (\phi \to \mathit{Result}) \to \mathit{Result},$$

for some type *Result* of 'final answers'. A function of this type receives both an input $x$ of type $\theta$ and a continuation $k$ of type $\phi \to \mathit{Result}$; instead of returning its result (of type $\phi$) directly, it instead calls the continuation $k$, passing the result to it, so that the value that is eventually returned is whatever final answer of type *Result* is computed by $k$. The basic monad of continuations is defined by

$$\textbf{type}\ M\,\alpha = (\alpha \to \mathit{Result}) \to \mathit{Result},$$

so that the function type shown above is $\theta \to M\,\phi$. There are well-known definitions of the operations *return* and $(\gg\!=)$ for this monad: $\mathit{return}\,x$ calls its continuation with the value $x$, and $p \gg\!= f$ calls $p$ with a continuation that, when applied to an argument $x$, calls $f\,x$ in turn. Versions of these appear below.

We can begin to define the types needed for a continuation-based toolkit by introducing a type *ContPipe* with this shape, but parametrising both it and the *Result* type with the types $\iota$ and $o$ of the input and output channels. We use **newtype** so that it is possible to make *ContPipe* an instance of various type classes.

$$\begin{aligned}&\textbf{newtype}\ \mathit{ContPipe}\,\iota\,o\,\alpha = \\ &\quad \mathit{MakePipe}\,\{\,\mathit{runPipe} :: \mathit{Cont}\,\iota\,o\,\alpha \to \mathit{Result}\,\iota\,o\,\}\end{aligned}$$

$$\textbf{type}\ \mathit{Cont}\,\iota\,o\,\alpha = \alpha \to \mathit{Result}\,\iota\,o$$

The notation $\{\,\mathit{runPipe} :: \ldots\,\}$ is a Haskell idiom for defining an inverse *runPipe* to the constructor *MakePipe*; both are elided at run-time. For clarity, we have also named the type $\mathit{Cont}\,\iota\,o\,\alpha$ of continuations expecting a value of type $\alpha$. This definition gives us enough structure to make $\mathit{ContPipe}\,\iota\,o$ into a monad for fixed types $\iota$ and $o$, in a way that precisely matches the basic monad of continuations.

$$\begin{aligned}&\textbf{instance}\ \mathit{Monad}\,(\mathit{ContPipe}\,\iota\,o)\ \textbf{where} \\ &\quad \mathit{return}\,x = \mathit{MakePipe}\,(\lambda\,k \to k\,x) \qquad (7)\\ &\quad p \gg\!= f = \qquad (8)\\ &\qquad \mathit{MakePipe}\,(\lambda\,k \to \mathit{runPipe}\,p\,(\lambda\,x \to \mathit{runPipe}\,(f\,x)\,k))\end{aligned}$$

In order to implement a pipeline toolkit, however, we must give each process more than a single continuation that it calls when it terminates. As indicated, we shall use three continuations instead of one: for a process of type $\mathit{ContPipe}\,\iota\,o\,\alpha$,

- the normal continuation, of type $\mathit{Cont}\,\iota\,o\,\alpha$, is called when the process terminates. This has been introduced above.
- the input continuation, of type $\mathit{InCont}\,\iota$, is called when the process stops to request input.
- the output continuation, of type $\mathit{OutCont}\,o$, is called when the process produces an output.

We can accommodate this by revealing that the type $\mathit{Result}\,\iota\,o$ actually has some internal structure, defining it in terms of another type *Answer* that we will fill in (much) later.

$$\textbf{type}\ \mathit{Result}\,\iota\,o = \mathit{InCont}\,\iota \to \mathit{OutCont}\,o \to \mathit{Answer}$$

Because the *Result* type appears twice in the definition of *ContPipe* – once directly, and once in the definition of *Cont* – this definition implies that not only will a process of type *ContPipe* receive these two extra continuations, but it will also pass on similar ones when it terminates by calling its own normal continuation.

The types of the input and output continuations themselves present a small problem, because in activating an input continuation, we will want to pass to it the output continuation that it should use when the input is ready; and in activating an output continuation, we must give it an input continuation to use when more input is needed. This makes us sail close to the maelstrom of impredicativity, defining two types that depend on each other in a way that no finite types can. Luckily **newtype** comes to the rescue, and we can make the following definitions.

```
newtype InCont ι =
  MakeInCont { resumeIn :: OutCont ι → Answer }

newtype OutCont o =
  MakeOutCont { resumeOut :: o → InCont o → Answer }
```

The usual inverses of the constructors are given the names *resumeIn* and *resumeOut* because they will be used to resume processes that have been suspended.

Dual to these are two functions that take a continuation – representing a process that is suspending itself – and an input or output continuation, and produce a new input or output continuation that includes the process. This makes the input and output continuations in a chain of processes behave like a pair of stacks.

```
suspendIn :: Cont ι o ( ) → InCont ι → InCont o
suspendIn k ik = MakeInCont (λ ok → k ( ) ik ok)

suspendOut :: Cont ι o ι → OutCont o → OutCont ι
suspendOut k ok = MakeOutCont (λ v ik → k v ik ok)
```

The types are a bit tricky here: looking at the type of *suspendOut*, if we take a continuation $k$ of type $Cont\ \iota\ o\ \iota$, it represents a process of type $ContPipe\ \iota\ o$ caught at a moment when it wants to input a value of type $\iota$; we are about to invoke the input continuation $ik$ for the process, and to $ik$ we must pass an *output* continuation of type $OutCont\ \iota$, telling it what to do with a value that it outputs. We obtain this output continuation by combining the process state $k$ with the output continuation for the process, which has type $OutCont\ o$. The equation defining *suspendOut* gives the surprisingly simple recipe for doing this. Its companion *suspendIn* is very similar, but uses the unit value ( ) to reflect the fact that no data passes from right to left in a pipeline.

We are now ready to implement the operations *input*, *output* and ‖. The first two look simple and are dual to each other. To input, a process suspends itself and resumes the process to its left, and to output a value, a process suspends itself and resumes the process to its right, passing it the output value as an argument.

```
instance PipeKit ContPipe where
  input =                                                        (9)
    MakePipe (λ k ik ok → resumeIn ik (suspendOut k ok))

  output v =                                                     (10)
    MakePipe (λ k ik ok → resumeOut ok v (suspendIn k ik))
```

In the implementation of *input*, we use *suspendOut* to produce an output continuation for the process to the left, and in *output* we use *suspendIn* to produce an input continuation for the process to the right.

Forming the process $p \parallel q$ amounts to calling $q$, but passing it an input continuation where an activation of $p$ has been created and immediately suspended.

```
instance PipeKit ContPipe where ...
  p ‖ q =                                                              (11)
    MakePipe (λ k ik ok →
      runPipe q k₉ (suspendIn (λ ( ) → runPipe p k₉) ik) ok)
    where k₉ _ _ _ = error "terminated"
```

In accordance with our conventions, both processes receive a normal continuation $k_9$ that barks an error message if either one ever terminates.

We could have set up the whole kit of parts in a way that was polymorphic in the answer type, adding an extra type parameter $\omega$ to each type constructor. Instead, we left the type *Answer* as a type still to be defined, and for programs that interact with the I/O system, it's appropriate to take *Answer* to be *IO* ( ). That lets us define a function *runContPipe* that connects a pipeline with appropriate input and output continuations at the two ends.

```
type Answer = IO ( )

runContPipe :: (Read ι, Show o) ⇒ ContPipe ι o ( ) → IO ( )
runContPipe p =
  runPipe p (λ ( ) _ _ → skip) ik ok
  where
    ik = MakeInCont (λ ok → (do x ← readLn; resumeOut ok x ik))
    ok = MakeOutCont (λ v ik → (do print v; resumeIn ik ok))
```

These definitions can also be generalised in a different way by replacing the *IO* monad by an arbitrary monad $m$ and recasting the whole construction as a monad transformer.

The operations *exit* and *effect* are easy to define in this implementation. Significantly, *exit* does not call any of its three continuations, and *effect* performs a top-level I/O action before calling its continuation to resume execution of the calling process.

```
instance PipeKit ContPipe where ...
  exit :: ContPipe ι o α
  exit = MakePipe (λ k ik ok → skip)

  effect :: IO α → ContPipe ι o α
  effect e = MakePipe (λ k ik ok → (do x ← e; k x ik ok))
```

If we make *ContPipe* into a monad transformer, then *effect* is the *lift* operation.

## 5 EVALUATION

The two implementations are interchangeable, in that the sort of programs written in terms of the *PipeKit* interface in Section 2 can be run using either one, simply by using *runDirPipe* or *runContPipe* as appropriate:

```
> runDirPipe (primes 1000)
2
3
5
...
7919
```

Exactly identical results are obtained if we use *runContPipe* instead.

Table 1. Timings for *primes n* (seconds)

| *n* | *Lazy streams* | *DirectPipe* | *ContPipe* | *HalfPipe* |
|---|---|---|---|---|
| 2000 | 0.07 | 0.18 | 0.06 | 0.11 |
| 3000 | 0.18 | 0.47 | 0.14 | 0.27 |
| 5000 | 0.55 | 1.56 | 0.43 | 0.85 |
| 7000 | 1.17 | 3.65 | 0.92 | 1.84 |
| 10000 | 2.71 | 9.19 | 2.13 | 4.21 |
| 15000 | 8.94 | 26.38 | 6.08 | 12.79 |
| 20000 | 21.51 | 57.05 | 13.49 | 31.09 |

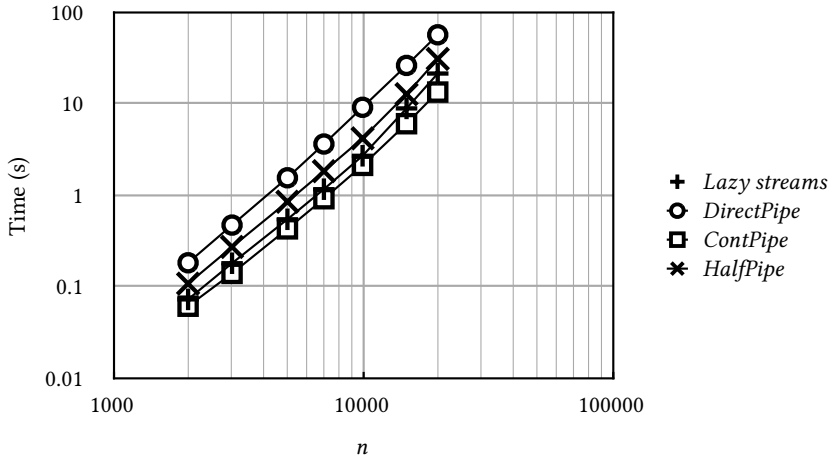


Fig. 1. Timings for *primes n*

There is little point in doing detailed timing studies on programs that are so simple, but it is worthwhile to get some idea of relative speeds. Each data point in Table 1 was obtained by averaging 20 runs of the *primes* program on a machine with an Intel Core i5 processor running at 1.8GHz and 8GB of memory. Compilation was with `ghc -O2`. In each case, output was redirected to `/dev/null`. In the table, each column shows times in seconds for the program *primes n* run with a range of values of *n*. The same data is shown as a log–log plot in Figure 1. For reference, the first column (marked 'Lazy') shows times for a program that uses the same algorithm but is based on lazy lists. The second and third columns correspond to the direct and continuation-based implementations shown in Sections 3 and 4 respectively. For completeness, the final column shows another implementation *HalfPipe* that uses the continuation-based transformation described by Voigtländer [2008] to improve the order of growth for sequential composition ≫= while leaving the parallel composition || unchanged. Details are given in the code archive accompanying this paper; a similar implementation has been used in recent versions of the *Conduits* package. Subsidiary experiments show that none of the running times are significantly affected by the indirection introduced by using each implementation via the type class *PipeKit*.

What the data reveal is that all four of these programs have running times that grow at about the same rate, but there is a more-or-less fixed ratio between their running times, with *DirectPipe* the slowest and *HalfPipe* rather faster. *ContPipe* and lazy streams are about equal in first place, faster than *DirectPipe* by a factor of 4 or more. In a real program, where a lot more work would

Table 2. Timings for *deepSeq n* and *deepPar n* (seconds)

| *n* | *DirectPipe seq* | *ContPipe seq* | *HalfPipe seq* | *DirectPipe par* | *ContPipe par* | *HalfPipe par* |
|---|---|---|---|---|---|---|
| 2000 | 0.09 | | | 0.08 | | 0.08 |
| 3000 | 0.24 | | | 0.24 | | 0.24 |
| 5000 | 0.80 | | | 0.77 | | 0.78 |
| 7000 | 1.81 | | | 1.73 | | 1.73 |
| 10000 | 5.66 | | | 5.44 | | 5.53 |
| 15000 | 19.56 | | | 18.78 | | 19.05 |
| 50000 | | 0.02 | 0.02 | | 0.02 | |
| 100000 | | 0.04 | 0.03 | | 0.04 | |
| 200000 | | 0.08 | 0.08 | | 0.08 | |
| 500000 | | 0.20 | 0.20 | | 0.20 | |
| 1000000 | | 0.39 | 0.41 | | 0.40 | |
| 2000000 | | 0.80 | 0.85 | | 0.82 | |
| 5000000 | | 1.99 | 2.12 | | 2.08 | |
| 10000000 | | 4.06 | 4.07 | | 4.22 | |

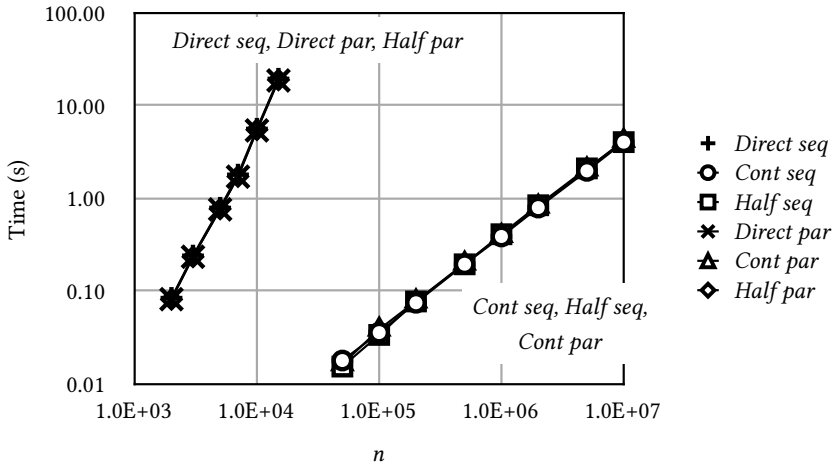


Fig. 2. Timings for *deepSeq n* and *deepPar n*

be done by the processes between communication events, and the running time might well be dominated by I/O, the overhead of either implementation would be acceptable. It would be a shame, though, to adopt a slow implementation needlessly, and so make pipelines less practical in more communication-intensive applications.

Another set of interesting timing results come from the programs

*deepSeq n* || *Pipe.take n*

*deepPar n* || *Pipe.take n*,

deliberately written to show up a difference in order of growth between the two implementations, with *deepSeq* and *deepPar* defined so that

$$deepSeq\, n = (\ldots((forever\,(output\ 0) \gg skip) \gg skip) \gg \ldots) \gg skip$$

$$deepPar\, n = skip \parallel (skip \parallel (\ldots \parallel (skip \parallel forever\,(output\ 0))\ldots)),$$

with $n$ occurrences of *skip* in each case. The process *deepSeq n* is the result of taking a process that forever outputs 0 and nesting it deeply inside instances of the monadic bind operator, whereas the process *deepPar n* takes the same process and makes is the *last* element of a pipeline of length $n + 1$. We might hope that each of these processes would take a time linear in $n$ to set up, and that each of the $n$ values then sent to the following *Pipe.take* process would be transmitted in a constant time, so that the whole program would take a time linear in $n$.

Table 2 shows the results of running these programs under the three implementations *DirectPipe*, *ContPipe* and *HalfPipe*, and the same data is plotted on log–log axes in Figure 2. It is immediately apparent that the tests fall into two groups that are orders of magnitude apart in performance, but with the results within each group closely consistent with each other. The different slopes of the lines indicate that the fast group shows linear growth, but the slow group shows an order of growth that is quadratic or worse. In point of fact, the behaviour observed shows closer to cubic than quadratic growth, and we suspect that this is caused by the program becoming dominated by garbage collection times that are proportional to the live space occupied, introducing another linear factor into the running time. Heap measurements show that the amount of storage allocated in these programs is roughly proportional to $n^2$ as expected.

The direct implementation *DirectPipe* is in the slow group for both sequential and parallel composition; *HalfPipe* is in the fast group for sequential composition, but remains slow for parallel composition; while only *ContPipe* achieves a linear order of growth for both. The reason for the poor performance of *DirectPipe* is is not hard to find, for when the process $q$ in $p \parallel q$ performs output, the relevant case in the definition of $\parallel$ is this one:

$$p \parallel q = \\ \quad \textbf{case}\ q\ \textbf{of}\ \ldots;\ Output\ v\ r \rightarrow Output\ v\ (p \parallel r);\ \ldots$$

Here we see that the value of $q$ is matched against a pattern, and a new *Output* value is constructed, with another invocation of $\parallel$ inside it. The new *Output* value is passed to the caller, also an invocation of $\parallel$, where the same pattern matching and construction happens again, until we reach the pipe connecting *deepPar* with *Pipe.take*, where the relevant case is found in $\parallel'$ instead of $\parallel$. The upshot is that the total cost of each communication is proportional to the depth of nesting, and the whole program will have at best a quadratic behaviour.

For *ContPipe*, no such slowdown is apparent. Indeed, we can see from the definition of $\parallel$ that both $p \parallel q$ and $q$ share the same output continuation, and there are no layers of interpretive code intervening between the process that outputs the zeroes and the *Pipe.take* process that is consuming them.

## 6 RELATING THE IMPLEMENTATIONS

In Section 3 we introduced a *PipeKit* instance called *DirectPipe*; later, in Section 4, we introduced a second instance called *ContPipe*. We now relate the two by defining a function

$$rep :: DirectPipe\ \iota\ o\ \alpha \rightarrow ContPipe\ \iota\ o\ \alpha$$

by interpreting each constructor for *DirectPipe* as an instruction to call one of the three continuations expected by a process of type *ContPipe*:

$$rep\,(Input\ h) = MakePipe\,(\lambda\, k\ ik\ ok \rightarrow \\ \quad \textbf{let}\ k'\ v = runPipe\,(rep\,(h\ v))\ k\ \textbf{in} \\ \quad resumeIn\ ik\,(suspendOut\ k'\ ok)) \tag{12}$$

$$rep\,(Output\,v\,r) = MakePipe\,(\lambda\,k\;ik\;ok \to \\ \quad \textbf{let}\;k'\,() = runPipe\,(rep\,r)\,k\;\textbf{in} \\ \quad resumeOut\;ok\;v\;(suspendIn\;k'\;ik)) \tag{13}$$

$$rep\,(Done\,x) = MakePipe\,(\lambda\,k \to k\,x) \tag{14}$$

Two ideas are involved in the clauses for *Input* and *Output*: first, that a recursive call of *rep* allows the residual action $h$ or $p$ to be composed with the continuation $k$ to form a new continuation; and second, the idea of suspending and resuming processes that is part of the continuation-based implementation of *input* and *output*.

We will now take as given the definitions of *PipeKit* operations on *DirectPipe*, and (forgetting for the moment the definitions given in Section 4) attempt to find appropriate definitions of the operations on *ContPipe* that are related to the given definitions according to *rep*, hoping to discover afresh the same definitions we gave earlier. Informally, we can say that these definitions should be chosen in such a way that *rep* is a homomorphism, so that we could apply *rep* to a whole program written in direct style, and use the relationship between *rep* and the operations to drive invocations of *rep* inwards until the whole program is written in continuation-passing style.

Let's agree to write subscripts $()_1$ and $()_2$ to identify the two implementations, so that

$$return_1 :: \alpha \to DirectPipe\,\iota\,o\,\alpha$$

$$return_2 :: \alpha \to ContPipe\,\iota\,o\,\alpha$$

The relation we seek between these two is that

$$rep\,(return_1\,x) = return_2\,x, \tag{15}$$

or in short that $rep \cdot return_1 = return_2$. Similarly, the two implementations of $\gg\!=$ are related by

$$rep\,(p \gg\!=_1 (\lambda\,x \to q)) = rep\,p \gg\!=_2 (\lambda\,x \to rep\,q), \tag{16}$$

or in short $rep\,(p \gg\!=_1 f) = rep\,p \gg\!=_2 (rep \cdot f)$. Together, (15) and (16) amount to saying that *rep* is a morphism of monads. Letting the types guide us, we can complete the picture by writing down the equations,

$$rep\;input_1 = input_2, \tag{17}$$

$$rep \cdot output_1 = output_2, \tag{18}$$

$$rep\,(p \parallel_1 q) = rep\,p \parallel_2 rep\,q. \tag{19}$$

We are now looking for implementations of $return_2$, etc., that satisfy these equations. In order to keep the calculations manageable, we can elide the **newtype** construction in *ContPipe* together with the constructor *MakePipe* and its inverse *runPipe*. When the derivation is finished, we can put them back in again, and the Haskell type checker will surely help us to do that correctly.

*Primitive operations.* For the primitive operations, the implementation task is easy. For example, equation (1) tells us that $return_1\,x = Done\,x$, so we can calculate from equations (15) and (14),

$$return_2\,x\,k = rep\,(Done\,x)\,k = k\,x,$$

and this agrees with equation (7). Again, equations (1) and (3) together imply that $input_1 = Input\;Done$, and so (appealing to equations (17), (12) and (14) and observing that in the following $k' = k$),

$$\begin{aligned} input_2\;k\;ik\;ok &= rep\,(Input\;Done)\;k\;ik\;ok \\ &= \textbf{let}\;k'\,x = rep\,(Done\,x)\,k\;\textbf{in}\;resumeIn\;ik\,(suspendOut\;k'\;ok) \\ &= resumeIn\;ik\,(suspendOut\;k\;ok). \end{aligned}$$

This justifies the definition (9). The story for *output* is similar: using equations (1) and (4) to obtain $output_1\ v = Output\ v\ (Done\ ())$ and applying (13) and (14),

$$\begin{aligned} output_2\ v\ k\ ik\ ok &= rep\ (Output\ v\ (Done\ ())) \\ &= \mathbf{let}\ k'\ () = rep\ (Done\ ())\ k\ \mathbf{in}\ resumeOut\ ok\ v\ (suspendIn\ k'\ ik) \\ &= resumeOut\ ok\ v\ (suspendIn\ k\ ik). \end{aligned}$$

This agrees with definition (10).

Now we should turn attention to the combinators $\gg\!=$ and $\|$. Both these are defined in the direct model by case analysis on one of their arguments, and we seek a continuation-based implementation that does no case analysis. This makes it appropriate to use an inductive argument for each operation, with an appeal ultimately to fixpoint induction. The argument we shall give applies to a single collection of processes arranged in a pipeline, and does not take into account the possibility, say, that the values transmitted between processes will themselves be processes, as is entirely possible. To make the entire argument rigorous, one would have to construct logical relations in the manner presented with exemplary clarity by Wand and Vaillancourt [2004].

*Sequential composition.* As regards $\gg\!=$, suppose $p_2 = rep\ p_1$ and $f_2\ x = rep\ (f_1\ x)$; we consider three cases, and show in each case that

$$rep\ (p_1 \gg\!=_1 f_1) = p_2\ (\lambda x \to f_2\ x\ k).$$

This justifies taking the right-hand side as the definition of $(p_2 \gg\!=_2 f_2)\ k$, consistent with equation (8).

- If $p_1 = Input\ h$ then using equations (2) and (12) gives

  $$\begin{aligned} rep\ (p_1 \gg\!=_1 f_1)\ k\ ik\ ok &= rep\ (Input\ (\lambda v \to h\ v \gg\!=_1 f_1))\ k\ ik\ ok \\ &= resumeIn\ ik\ (suspendOut\ k'\ ok), \end{aligned}$$

  where $k'\ v = rep\ (h\ v \gg\!=_1 f_1)\ k$. But according to equation (12), if $p_2 = rep\ p_1$ then

  $$p_2\ (\lambda x \to f_2\ x\ k)\ ik\ ok = resumeIn\ ik\ (suspendOut\ k''\ ok),$$

  where $k''\ v = rep\ (h\ v)\ (\lambda x \to f_2\ x\ k)$. We may inductively assume that $k'$ and $k''$ agree.
- If $p_1 = Output\ v\ r_1$ then using equations (2) and (13) gives

  $$\begin{aligned} rep\ (p_1 \gg\!=_1 f_1)\ k\ ik\ ok &= rep\ (Output\ v\ (r_1 \gg\!=_1 f_1))\ k\ ik\ ok \\ &= resumeOut\ ok\ v\ (suspendIn\ k'\ ik), \end{aligned}$$

  where $k'\ () = rep\ (r_1 \gg\!=_1 f_1)\ k$. Again, according to equation (13), if $p_2 = rep\ p_1$ then

  $$\begin{aligned} &p_2\ (\lambda x \to f_2\ x\ k)\ ik\ ok \\ &\quad = resumeOut\ ok\ y\ (suspendIn\ k''\ ik), \end{aligned}$$

  where $k''\ () = rep\ q\ (\lambda x \to f_2\ x\ k)$. As before, we may assume that $k'$ and $k''$ agree.
- If $p_1 = Done\ x$, then equation (12) gives $p_2\ k = k\ x$ and, using equation (2), we obtain

  $$rep\ (p_1 \gg\!=_1 f_1)\ k = rep\ (f_1\ x)\ k = f_2\ x\ k = p_2\ (\lambda x \to f_2\ x\ k).$$

Together, these results motivate the definition of $\gg\!=_2$ in Section 4.

*Parallel composition.* Turning now to parallel composition, the direct implementation uses two mutually recursive functions named $\|_1$ and $\|_1'$, and to make an inductive argument, we must formulate hypotheses about both of them. Recall that in $p_1 \|_1 q_1$, it is $q_1$ that is active, whereas in $p_1 \|_1' h_1$, the right-hand process has evolved into a handler function $h_1$ that is waiting for input, and $p_1$ has become active in its place, until such a time as it produces output. If $p_2 = rep\ p_1$ and $q_2 = rep\ q_1$ and $h_2\ v = rep\ (h_1\ v)$, we formulate the hypotheses as follows.

$$rep\,(p_1 \parallel_1 q_1)\,k\;ik\;ok = q_2\,k_9\,(suspendIn\,(\lambda\,() \to p_2\,k_9)\;ik)\;ok, \tag{20}$$

$$rep\,(p_1 \parallel_1' h_1)\,k\;ik\;ok = p_2\,k_9\;ik\,(suspendOut\,(\lambda\,v \to h_2\,v\,k_9)\;ok). \tag{21}$$

In both equations, $k_9$ is the error-signalling continuation introduced in the definition of $\parallel_2$ (equation (11)). Now again we reason by cases, examining the form of $q_1$ in our analysis of $p_1 \parallel_1 q_1$ and the form of $p_1$ in analysing $p_1 \parallel_1' h_1$. For $p_1 \parallel_1 q_1$, two cases arise:

- If $q_1 = Input\,h_1$, then $p_1 \parallel_1 q_1 = p_1 \parallel_1' h_1$. Let $p_2 = rep\,p_1$ and $q_2 = rep\,q_1$ and $h_2\,v = rep\,(h_1\,v)$. We can expand the left hand side of (20) as follows and apply induction hypothesis (21) to $p_1 \parallel_1' h_1$.

  $rep\,(p_1 \parallel_1 q_1)\,k\;ik\;ok = rep\,(p_1 \parallel_1' h_1)\,k\;ik\;ok$
  $\quad = p_2\,k_9\;ik\,(suspendOut\,(\lambda\,v \to h_2\,v\,k_9)\;ok).$

  Now we use (12) on the right hand side of (20), since $q_2 = rep\,(Input\,h_1)$.

  $q_2\,k_9\,(suspendIn\,(\lambda\,() \to p_2\,k_9)\;ik)\;ok$
  $\quad = resumeIn\,(suspendIn\,(\lambda\,() \to p_2\,k_9)\;ik)\,(suspendOut\,(\lambda\,v \to h_2\,v\,k_9)\;ok)$
  $\quad = p_2\,k_9\;ik\,(suspendOut\,(\lambda\,v \to h_2\,v\,k_9)\;ok).$

  The two sides agree.
- If $q_1 = Output\,v\,r_1$, then $p_1 \parallel_1 q_1 = Output\,v\,(p_1 \parallel_1 r_1)$. Let $p_2 = rep\,p_1$ and $q_2 = rep\,q_1$ and $r_2 = rep\,r_1$. Expanding the left hand side of (20) we obtain,

  $rep\,(p_1 \parallel_1 q_1)\,k\;ik\;ok = rep\,(Output\,v\,(p_1 \parallel_1 r_1))\,k\;ik\;ok$
  $\quad = resumeOut\;ok\,v\,(suspendIn\,(\lambda\,() \to rep\,(p_1 \parallel_1 r_1)\,k)\;ik).$

  Since $q_2 = rep\,(Output\,v\,r_1)$, expanding the right hand side of (20) with (13) gives,

  $q_2\,k_9\,(suspendIn\,(\lambda\,() \to p_2\,k_9)\;ik)\;ok$
  $\quad = resumeOut\;ok\,v\,(suspendIn\,(\lambda\,() \to r_2\,k_9)\,(suspendIn\,(\lambda\,() \to p_2\,k_9)\;ik)).$

  We have reached two expressions that differ only in the way the suspended input continuation is formed. But if $ok'$ is any output continuation, we find

  $resumeIn\,(suspendIn\,(\lambda\,() \to rep\,(p_1 \parallel_1 r_1)\,k)\;ik)\;ok' = rep\,(p_1 \parallel_1 r_1)\,k\;ik\;ok',$

  and

  $resumeIn\,(suspendIn\,(\lambda\,() \to r_2\,k_9)\,(suspendIn\,(\lambda\,() \to p_2\,k_9)\;ik))\;ok'$
  $\quad = r_2\,k_9\,(suspendIn\,(\lambda\,() \to p_2\,k_9)\;ik)\;ok'.$

  We may assume as an instance of induction hypothesis (20) that these two are equal, and since *resumeIn* is injective, the two suspensions are equal also.

This completes the reasoning for $p_1 \parallel_1 q_1$. For $p_1 \parallel_1' h_1$, again there are two cases, depending on the value of $p_1$.

- If $p_1 = Input\,g_1$, then $p_1 \parallel_1' h_1 = Input\,(\lambda\,w \to g_1\,w \parallel_1' h_1)$. Putting $p_2 = rep\,p_1$ and $g_2\,w = rep\,(g_1\,w)$, we can expand the left hand side of (21) as follows.

  $rep\,(p_1 \parallel_1' h_1)\,k\;ik\;ok = rep\,(Input\,(\lambda\,w \to g_1\,w \parallel_1' h_1))\,k\;ik\;ok$
  $\quad = resumeIn\;ik\,(suspendOut\,(\lambda\,w \to rep\,(g_1\,w \parallel_1' h_1)\,k)\;ok).$

  Using (12) on the right hand side of (21) gives, since $p_2 = rep\,(Input\,g_1)$,

  $p_2\,k_9\;ik\,(suspendOut\,(\lambda\,v \to h_2\,v\,k_9)\;ok)$
  $\quad = resumeIn\;ik\,(suspendOut\,(\lambda\,w \to g_2\,w\,k)\,(suspendOut\,(\lambda\,v \to h_2\,v\,k_9)\;ok).$

As in the second case in the previous argument, we must compare two suspensions, and a similar argument shows that they are equal.

- If $p_1 = \mathit{Output}\, v\, r_1$, then $p_1 \parallel_1' h_1 = r_1 \parallel_1 h_1\, v$. Expanding the left hand side of (21) and applying an induction hypothesis for $r_1 \parallel_1 h_1\, v$ gives, since $p_2 = \mathit{rep}\,(\mathit{Output}\, v\, r_1)$,

$$\begin{aligned}&\mathit{rep}\,(p_1 \parallel_1' h_1)\, k\ ik\ ok = \mathit{rep}\,(r_1 \parallel_1 h_1\, v)\, k\ ik\ ok\\ &\quad = h_2\, v\, k_9\, (\mathit{suspendIn}\,(\lambda\,() \to r_2\, k_9)\ ik)\ ok.\end{aligned}$$

Now using (13) on the right hand side of (21) gives,

$$\begin{aligned}&p_2\, k_9\ ik\,(\mathit{suspendOut}\,(\lambda\, v \to h_2\, v\, k_9)\ ok)\\ &\quad = \mathit{resumeOut}\,(\mathit{suspendOut}\,(\lambda\, v \to h_2\, v\, k_9)\ ok)\,(\mathit{suspendIn}\,(\lambda\,() \to r_2\, k_9)\ ik)\\ &\quad = h_2\, v\, k_9\,(\mathit{suspendIn}\,(\lambda\,() \to r_2\, k_9)\ ik).\end{aligned}$$

Again, the two sides agree.

This completes the inductive proof of (20) and (21), and justifies taking the right hand side of (20) as the definition of $p_2 \parallel_2 q_2$, as we did in equation (11). With this result, we have completed a derivation of the continuation-based toolkit of Section 4 from the direct-style toolkit. The information that we had to inject is given by the type definitions and the function *rep*, which draws in definitions of *suspendIn* and *suspendOut*.

## 7 FORKS AND JOINS

The *Conduit* library supports, in addition to straight pipelines, constructions that allow pipelines to be grafted together at their inputs or at their outputs. Similar extensions can be made to the continuation-based pipeline toolkit. If $p$ and $q$ are two pipelines, then $\mathit{join}\, p\, q$ is a pipeline that synchronises the outputs of $p$ and $q$, outputting the pair $(x, y)$ whenever $p$ outputs $x$ and $q$ outputs $y$.

A simple definition of *join* assumes that $p$ and $q$ never input or terminate. Its heart is a loop, expressed with tail recursion, that runs $p$ and then $q$ until both are ready to output, then outputs a pair of values and repeats.

```
join₀ :: ContPipe () o () → ContPipe () η () → ContPipe () (o, η) ()
join₀ p q =
  MkPipe (λ k ik ok →
    loop (MkInCont (runPipe p k₉ ik₉)) (MkInCont (runPipe q k₉ ik₉)) ok)
  where
    loop ik₁ ik₂ ok =
      resume_in ik₁ (MkOutCont (λ x ik₁' →
        resume_in ik₂ (MkOutCont (λ y ik₂' →
          resume_out ok (x, y) (MkInCont (λ ok' →
            loop ik₁' ik₂' ok'))))))

    ik₉ = MkInCont (λ _ → error "input forbidden in join")
```

As an example, we could form the process $\mathit{join}\,(\mathit{upfrom}\,1)\,(\mathit{primes}\, n)$ that outputs the first $n$ primes, labelling each one with its serial number, before stopping with *exit*.

With care, we can generalise *join* so that input actions by the two processes are interleaved, and each may terminate with a value that becomes the value of the whole construct. Because the input channel is shared by the two processes, the left context of the processes must be held in a way that is external to both of them, and a simple way to do this uses a reference cell in the *IO* monad. Treating the right context in a similar way allows for termination also.

```
join :: ContPipe ι o α → ContPipe ι η α → ContPipe ι (o, η) α
join p q =
  MkPipe (λ k ik ok → do ir ← newIORef ik; or ← newIORef ok; body k ir or)
  where
    body k ir or =
      loop (MkInCont (runPipe p k₀ ik₀)) (MkInCont (runPipe q k₀ ik₀))
      where
        loop ik₁ ik₂ =
          resume_in ik₁ (MkOutCont (λ x ik₁' →
            resume_in ik₂ (MkOutCont (λ y ik₂' →
              do ok ← readIORef or;
                 resume_out ok (x, y) (MkInCont (λ ok' →
                   do writeIORef or ok'; loop ik₁' ik₂'))))))

        k₀ x _ _ =
          do ik ← readIORef ir; ok ← readIORef or; k x ik ok

        ik₀ =
          MkInCont (λ ok' →
            do ik ← readIORef ir; resume_in ik (MkOutCont (λ x ik' →
                 do writeIORef ir ik'; resume_out ok' x ik₀)))
```

The arbitrary interleaving of input makes this generalisation perhaps less useful than it might appear, but it does show what can be done. The method used here to provide for termination can be copied for ordinary composition of pipes, but at the cost of monitoring the input and output channels in a way that would again spoil the order of growth for deeply nested compositions.

Dual to *join* is an operation *fork* that takes two processes and synchronises them on their inputs, feeding the same input value to both and interleaving their outputs. This time the two processes are first run until both are ready to input, and the driver loop then repeatedly obtains an item of input and feeds it first to one and then to the other process.

```
fork :: ContPipe ι o () → ContPipe ι o () → ContPipe ι o ()
fork p q =
  MkPipe (λ k ik ok → (do or ← newIORef ok; body ik or))
  where
    body ik or =
      runPipe p k₉ (MkInCont (λ ok₁ →
        runPipe q k₉ (MkInCont (λ ok₂ →
          loop ik ok₁ ok₂)) ok₀)) ok₀

      where
        loop ik ok₁ ok₂ =
          resume_in ik (MkOutCont (λ v ik' →
            resume_out ok₁ v (MkInCont (λ ok₁' →
              resume_out ok₂ v (MkInCont (λ ok₂' → loop ik' ok₁' ok₂'))))))

        ok₀ =
          MkOutCont (λ v ik' →
            do ok ← readIORef or; resume_out ok v (MkInCont (λ ok' →
                 do writeIORef or ok'; resume_in ik' ok₀)))
```

We do not provide for termination of the processes here, but the definition of *fork* can be extended to that when one of the processes has terminated, the other one is also run to completion, and the construct then yields the two return values as a pair. We will not try the reader's patience by including the code for this.

## 8 RELATED WORK

Our implementations of pipes share broadly the same interface as the *Pipes* and *Conduit* libraries. These evolved from an earlier library called *Iteratee* due to Kiselyov [2012] that implemented a different interface, with an explicit command corresponding to our *input*, but with *output* replaced by having a process return the next value to be transmitted. These conventions make it more difficult to write processes with an internal control state that persists between one output and the next, such as the *sieve* process in the primes program. They also require that processes with an internal data state use the *State* monad explicitly rather than the natural recursive scheme exemplified by our process *upfrom*. In these respects, the later libraries represent an advance in convenience over *Iteratee*, but where they have gained in convenience, they have arguably lost in efficiency, for *Iteratee* has an alternative implementation in CPS style that is reportedly faster; this paper corrects that deficiency.

The use of functional programming to simulate parallel processes has a long history, and it would be surprising if similar ideas had not been explored in the past. An important milestone is the *Fudgets* library of Carlsson and Hallgren [1993], a framework for GUI programming in Haskell that is based on 'stream processors', closely resembling our pipelines. The representation of stream processors and the implementation of what the authors call 'serial composition' closely resemble the direct-style representation of pipelines and implementation of parallel composition in our Section 3. They also describe an operation that they call 'parallel composition' that resembles the *fork* combinator in our Section 7.

Interestingly, Claessen [2004] describes a library of parser combinators that uses a type very similar to *DirectPipe*, but implements disjunction of parsers by an operation like *fork* that collects the results produced by both operands, gaining efficiency by exploring the search space generated by the grammar in a breadth-first instead of depth-first manner.

These implementations, like *DirectPipe* and the *Pipes* and *Conduits* libraries, make use of continuations in a local way, because the argument of *Input* is a function to be invoked when input is ready, or in common terminology, a *callback*: it encodes the future of the process that invokes the *input* action. But this is different from true continuation-passing style, which demands that all calls are in tail position, and all functions return the final *Answer* type, so a continuation encodes the entire future of the computation.

The use of continuation-passing style to encode lazy streams seems to be part of the folk-lore, and gives an alternative motivation for the definitions in Section 4. The type $Stream\,\alpha$ of infinite streams over $\alpha$ can be viewed as a solution to the domain equation

$$Stream\,\alpha \cong (\alpha, Stream\,\alpha).$$

Adopting continuation-passing style and currying, the above equation becomes

$$Stream\,\alpha \cong (\alpha \to Stream\,\alpha \to Answer) \to Answer$$

If we name the parenthesised, inner type $StreamCont\,\alpha$, we reach the pair of equations,

$$Stream\,\alpha \cong StreamCont\,\alpha \to Answer$$
$$StreamCont\,\alpha \cong \alpha \to Stream\,\alpha \to Answer,$$

making an obvious connection with the types $InCont\,\alpha$ and $OutCont\,\alpha$ defined in Section 4. On this view, a process with one input channel and one output channel becomes a function $Stream\,\iota \to Stream\,o$ whose type expands to

$$Stream\,\iota \to StreamCont\,o \to Answer,$$

making the connection with *ContPipe* all the more apparent. Pseudo-parallel composition of pipes becomes composition of functions from streams to streams, and composition expressed in CPS gives the essence of the definition of $\|$ from Section 4.

Shivers and Might [2006] use exactly this representation of processes, and provide primitives identical with our *input*, *output* and $\|$, implementing them first in Scheme and then in SML with *callcc*. They consider the extent to which compiler optimisations can fuse processes together, removing needless communication between them. Against the background of this earlier work, the contribution of the present paper can be seen as showing that these ideas can be used to implement the monadic interface in Section 2, and showing that the direct and continuation-passing implementations are related by the algebraic reasoning in Section 6.

Hinze [2000] gives hints for deriving an efficient, continuation-passing implementation of backtracking, by discerning the most general context in which a process may run, and choosing the types of continuations accordingly. It is not clear whether the continuation-based implementation of coroutines given in this paper could be derived by a similar process of discernment, but the function *rep* does a similar job of revealing the relationship between the commands that may be invoked by a process and the continuations that form its context. A later paper [Hinze 2012] makes the connection between these programming ideas and concepts from Category Theory.

In Functional Reactive Programming, the dominant metaphor is time-dependent signals, conceived in the abstract as functions $Time \to \alpha$, but implemented as streams of discrete samples. Perez et al. [2016] have shown that the absolute measure of time may be treated independently of other aspects of the framework. They study *monadic stream functions*, with a type $MSF\,m\,\alpha\,\beta$ which may be defined for an arbitrary monad $m$ by

$$\begin{aligned}&\textbf{newtype}\ MSF\,m\,\alpha\,\beta = \\ &\quad MakeMSF\,\{\,step :: \alpha \to m\,(\beta, MSF\,m\,\alpha\,\beta)\,\}\end{aligned}$$

Via the monad $m$, it is possible to permit side effects and IO (by taking $m = IO$), but also to re-introduce time-dependence (by taking $m = ReaderT\ Time$).

There is a composition operator defined in essence by

$$\begin{aligned}&(\|) :: Monad\,m \Rightarrow MSF\,m\,\alpha\,\beta \to MSF\,m\,\beta\,\gamma \to MSF\,\alpha\,\gamma \\ &p \,\|\, q = \\ &\quad MakeMSF\,(\lambda\,x \to \\ &\qquad \textbf{do}\ (y, p') \leftarrow step\,p\,x; \\ &\qquad\qquad (z, q') \leftarrow step\,q\,y; return\,(z, p' \,\|\, q'))\end{aligned}$$

This reveals the synchronous, lock-step nature of the framework, since both $p$ and $q$ take a step in each step of $p \,\|\, q$. The one-to-one correspondence between inputs and outputs of a process that is natural in this framework contrasts with the one-to-many or many-to-one or indeed many-to-many relationships that are possible with asynchronous pipelines.

In such a system, the asymptotic slowdown shown by our version of pipelines cannot happen. However it is put together, a chain of $n$ processes will form a binary tree with $n - 1$ isntances of the $\|$ operator as internal nodes. If each process does at least a fixed amount of work in each step, and if the overhead introduced by $\|$ is bounded by a constant, then the overhead will be at most a fixed proportion of the work done. Note that the expression $p' \,\|\, q'$ appearing at the end of the definition

of || might reasonably be called a continuation, since it is a function representing the future of the process; but as before, that doesn't mean that continuation-passing style is being exploited here.

Other, earlier realisations of FRP such as Yampa [Hudak et al. 2002] offer similar composition operators, specialised to timed updates, that maintain the synchronous nature of the execution, and use a similar, continuation-based representation of the system state [Nilsson et al. 2002]. In these systems, it is fruitful to optimise the implementation in a different way, by having a representation of processes that identifies those processes known to be, for example, constant in value or a pure, stateless transformation, and having the combining forms treat them specially.

Netwire [Söylemez 2016] is another realisation of FRP that breaks the one-to-one correspondence between inputs and outputs by allowing components not to produce an output on some cycles, and also uses a continuation-based representation of system state. Nevertheless, the execution model remains synchronous, with missing values represented by replacing $\alpha$ with $Maybe\,\alpha$ where necessary. By way of contrast [van der Ploeg 2013] adopts an asynchronous model of execution for FRP, and represents reactive components for events $\epsilon$ by the type

$$\begin{aligned}&\textbf{data}\ React\,\epsilon\,\alpha = \\ &\quad Done\,\alpha \\ &\mid\ Await\,(Request\,\epsilon)\,(Occur\,\epsilon \longrightarrow React\,\epsilon\,\alpha),\end{aligned}$$

where both $Request\,\epsilon$ and $React\,\epsilon$ represent sets of events, with obvious connections to our direct implementation of processes. It would be interesting to investigate whether a CPS version is possible.

Lastly, it seems relevant to mention ReactiveX [2015], a library for reactive programming that is implemented for multiple languages. In ReactiveX, an *Observable* presents a stream of values using a callback (arguably equivalent to a continuation), but the interface is rather different and lacks the symmetry between input and output that is shown by the *Pipes* and *Conduit* libraries in Haskell.

## 9 CONCLUSION

We have studied an interface for creating process pipelines, and two implementations, one similar to the existing Haskell libraries, and another that uses continuations. Neither implementation is very much more complex than the other in terms of code size, though the type structure of the continuation-based implementation is a bit more intricate. For simple test programs where communication dominates, the continuation-based implementation is substantially faster, and sometimes faster even than the native lazy streams of Haskell. In addition, it is possible to find deeply nested programs of different kinds where the direct implementation suffers a slow-down that the continuation-based implementation avoids.

The difference between the two implementations can be understood in the following terms. When a process suspends itself for input or output in the direct implementation, the signal to suspend is noticed by each element in the chain of dynamic contexts leading to the point of suspension, and a data structure representing the suspended state is built at that point. If the program suspends many times in the same place, then the same data structure is built each time. The continuation-based implementation already has a representation of the execution context in the form of the current continuations, so there is no need to build a fresh one.

The impact of this slow-down in the case of sequential composition is well known, and there is a uniform technique for improving programs that are based on a 'free monad' by introducing continuations [Voigtländer 2008]. This technique explains the relationship between the two implementations of pipes given here, at least as regards sequential composition; for parallel composition, however, the continuation-based implementation does not appear to fall into this pattern, and an extra ingredient in the form of the mutually applicable input and output continuations

seems to be needed. With that ingredient, the derivation in Section 6 shows how the details of the implementation follow automatically.

For simplicity, we have forbidden processes from terminating when they are composed in parallel. As noted above, we can easily observe a convention that a process that terminates will forever output a sentinel value to indicate end-of-stream: but that in essence means that we forbid it from terminating at all. Alternatively, we could follow each pipeline component with a call of *exit*, so as to stop the whole program and return from the enclosing *runContPipe* call.

As a third possibility, we might want the convention that the first process to terminate in a pipeline of parallel processes immediately terminates the whole pipeline, but allows other parts of the program to continue. The difficulty raised by this is that the context of a process in our implementation consists only of its input and output continuations. This becomes problematic if, say process $p_3$ terminates in the program

$$p_1 \parallel ((p_2 \parallel p_3 \parallel p_4) \gg q) \parallel p_5.$$

In this case, termination of $p_3$ should cause immediate termination of the three-process pipeline of which it is part, and leave the process $q$ to run in a pipeline that also contains $p_1$ and $p_5$ in their current state. The problem is that, if the input and output continuations of $p_3$ are represented as simple functions, then we have no access to the states of $p_1$ and $p_5$ that are embedded within them. Even where the types permit, it is quite wrong to treat termination of $p_3$ by applying the continuation for $p_2 \parallel p_3 \parallel p_4$ to its current input and output continuations, for that would leave $q$ sandwiched between the remnants of $p_2$ and $p_4$.

It would be good to find out whether the behaviour proposed as correct can be made consistent with other desirable properties of the toolkit, whether an appropriate, concrete representation of continuations that permits inspection would allow it to be implemented, and what the effect would be on the execution speed of basic pipeline operations. It might also be fruitful to explore a design suggested by the stream processing language Ziria [Stewart et al. 2015], where the type system ensures that in a parallel composition $p \parallel q$ at most one of $p$ and $q$ may terminate with a result. We leave these questions for future investigation.

More widely, one can imagine any number of schemes where independent sequential processes communicate with others by sending messages, and it is natural to describe such protocols in an operational style using a free monad plus an interpreter for the messages. It would be interesting to study whether continuation-based implementations of such schemes could be derived in a systematic way that generalises the calculation done in this paper.

## ACKNOWLEDGMENTS

The author thanks Jeff Polakow for a fruitful correspondence about the representation of streams in CPS.

## REFERENCES

Magnus Carlsson and Thomas Hallgren. 1993. FUDGETS: A Graphical User Interface in a Lazy Functional Language. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*. ACM, New York, NY, USA, 321–330. https://doi.org/10.1145/165180.165228

Koen Claessen. 2004. Parallel Parsing Processes. *J. Funct. Program.* 14, 6 (2004), 741–757. https://doi.org/10.1017/S0956796804005192

Gabriel Gonzalez. 2012. Haskell *Pipes* library. (2012). http://hackage.haskell.org/package/pipes

Ralf Hinze. 2000. Deriving backtracking monad transformers. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, Martin Odersky and Philip Wadler (Eds.). ACM, 186–197. https://doi.org/10.1145/351240.351258

Ralf Hinze. 2012. Kan Extensions for Program Optimisation Or: Art and Dan Explain an Old Trick. In *Mathematics of Program Construction - 11th International Conference, MPC 2012, Madrid, Spain, June 25-27, 2012. Proceedings*, Jeremy Gibbons and Pablo Nogueira (Eds.). 324–362. https://doi.org/10.1007/978-3-642-31113-0_16

Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2002. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming, 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002, Revised Lectures*. 159–187. https://doi.org/10.1007/978-3-540-44833-4_6

Oleg Kiselyov. 2012. Iteratees. In *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings (Lecture Notes in Computer Science)*, Tom Schrijvers and Peter Thiemann (Eds.), Vol. 7294. Springer, 166–181. https://doi.org/10.1007/978-3-642-29822-6_15

Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 51–64. https://doi.org/10.1145/581690.581695

Ivan Perez, Manuel Bärenz, and Henrik Nilsson. 2016. Functional Reactive Programming, Refactored. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, New York, NY, USA, 33–44. https://doi.org/10.1145/2976002.2976010

ReactiveX. 2015. Documentation for *Observable*. (2015). http://reactivex.io/documentation/observable.html

Olin Shivers and Matthew Might. 2006. Continuations and Transducer Composition. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 295–307. https://doi.org/10.1145/1133981.1134016

Michael Snoyman. 2011. Haskell *Conduit* library. (2011). http://hackage.haskell.org/package/conduit

Ertugrul Söylemez. 2016. Haskell *Netwire* library. (2016). http://hackage.haskell.org/packages/netwire

Gordon Stewart, Mahanth Gowda, Geoffrey Mainland, Bozidar Radunovic, Dimitrios Vytiniotis, and Cristina Luengo Agullo. 2015. Ziria: A DSL for Wireless Systems Programming. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). 415–428. https://doi.org/10.1145/2694344.2694368

Valery Trifonov. 2003. Simulating quantified class constraints. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, August 28, 2003*. ACM, 98–102. https://doi.org/10.1145/871895.871906

Atze van der Ploeg. 2013. Monadic functional reactive programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*. 117–128. https://doi.org/10.1145/2503778.2503783

Janis Voigtländer. 2008. Asymptotic Improvement of Computations over Free Monads. In *Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008. Proceedings (Lecture Notes in Computer Science)*, Philippe Audebaud and Christine Paulin-Mohring (Eds.), Vol. 5133. Springer, 388–403. https://doi.org/10.1007/978-3-540-70594-9_20

Mitchell Wand and Dale Vaillancourt. 2004. Relating models of backtracking. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*, Chris Okasaki and Kathleen Fisher (Eds.). ACM, 54–65. https://doi.org/10.1145/1016850.1016861