JFP **30**, e22, 40 pages, 2020. © The Author(s), 2020. Published by Cambridge University Press
doi:10.1017/S0956796820000192

Faster coroutine pipelines: A reconstruction

RUBEN P. PIETERS AND TOM SCHRIJVERS

KU Leuven, Leuven, Belgium

(*e-mails:* `ruben.pieters@cs.kuleuven.be`, `tom.schrijvers@cs.kuleuven.be`)

Abstract

The three-continuation approach to coroutine pipelines efficiently represents a large number of connected components. Previous work in this area introduces this alternative encoding but does not shed much light on the underlying principles for deriving this encoding from its specification. This paper gives this missing insight by deriving the three-continuation encoding based on eliminating the mutual recursion in the definition of the connect operation. Using the same derivation steps, we are able to derive a similar encoding for a more general setting, namely bidirectional pipes. Additionally, we evaluate the encoding in an advertisement analytics benchmark where it is as performant as `pipes`, `conduit`, and `streamly`, which are other common Haskell stream processing libraries.

1 Introduction

Coroutine pipelines provide a compositional approach to processing streams of data that is both efficient in time and space, thanks to a targeted form of lazy evaluation interacting well with side-effects like I/O. Two prominent Haskell libraries for coroutine pipelines are `pipes` (Gonzalez, 2012) and `conduit` (Snoyman, 2011). Common to both libraries is their representation of pipelines by an algebraic data type (ADT). The `streamly` library (Kumar, 2017) is a more recent alternative for stream processing in Haskell, which focuses on modifying the stream directly as opposed to constructing stream transformers.

Spivey (2017) has recently presented a novel Haskell representation that is entirely function based. His representation is an adaptation of Shivers and Might's earlier three-continuation representation (Shivers & Might, 2006) and exhibits a very efficient *connect* operation for connecting pipes.

Spivey proves that his representation is equivalent to a simple ADT-based specification. Yet, neither his proof nor Shivers and Might's explanation sheds much light on the underlying principles used to come up with the efficient representation. This makes it difficult to adapt the representation to other settings.

This paper remedies the situation by systematically deriving the efficient function-based representation from the simple, but inefficient ADT-based representation. Our derivation consists of known transformations and constructions that are centered around folds with appropriate algebras. Our derivation clarifies the limitations of the efficient representation and enables us to derive a similarly efficient representation for the two-way pipes featured in the `pipes` library.

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

The specific contributions of this paper are

- We present a systematic derivation of Spivey's efficient representation starting from a simple executable specification. Our derivation only consists of known transformations, most of which concern structural recursion with folds and algebras. It also explains why the efficient representation only supports connecting "never-returning" pipes.
- We apply our derivation to a more general definition of pipes used by the `pipes` library, where the communication between adjacent pipes is bidirectional rather than unidirectional.
- Our benchmarks demonstrate that the connect operator for the bidirectional three-continuation approach improves upon the performance of the `pipes`, `conduit`, and `streamly` libraries. However the performance of other common operations, such as *map*, *filter*, and *fold*, is slower compared to other streaming libraries. Nevertheless, it is as performant as the conventional libraries in the advertisement analytics benchmark.
- We discuss how we ported several performance optimization techniques, inspired by the traditional stream processing libraries.

The rest of this paper is organized as follows. Section 2 briefly introduces both the ADT pipes encoding and the three-continuation approach. Section 3 derives the fast connecting operation for a simplified setting. Section 4 derives the fast connecting operation for the original pipe setting. Section 5 extends Spivey's approach with the bidirectional pipes operations. Section 6 presents the results of several benchmarks, comparing the encoding presented in this paper to several common Haskell stream processing libraries. Section 7 discusses related work and Section 8 concludes this paper.

This paper is based on an earlier publication: "Faster Coroutine Pipelines: A Reconstruction" (Pieters & Schrijvers, 2019). The main changes are the addition of examples and extended explanations in Sections 3 and 4, an overhaul of Section 5 with more extensive explanation and examples, and in Section 6 where an additional stream processing library `streamly` is added to the benchmarks, more information regarding the benchmarks is added, we investigate some additional use case-driven benchmarks and we discuss several performance optimization techniques applied on the continuation-based encoding.

2 Motivation

This section introduces the ADT pipes encoding and then contrasts it with the three-continuation encoding. This serves as both a background introduction and a motivation for a better understanding of the relation between both encodings.

2.1 Pipes

We start with a unidirectional version of the `pipes` library. A unidirectional *pipe* can receive i values, output o values, and return a values. Unidirectional pipes stand in contrast

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

to bidirectional pipes, which we cover in Section 5. We represent a unidirectional pipe as an abstract syntax tree where each node is an input, output, or return operation. This is expressed in Haskell with the following ADT:

```
data Pipe i o a = Input (i → Pipe i o a)
                | Output o (Pipe i o a)
                | Return a
```

This data type exhibits a monadic structure where the bind operation $(\gg=) :: (Pipe\ i\ o\ a) \to (a \to Pipe\ i\ o\ b) \to Pipe\ i\ o\ b$ replaces *Return* nodes containing a value a with the structure obtained by applying it to the function $a \to Pipe\ i\ o\ b$.

```
instance Monad (Pipe i o) where
  return = Return
  (Input h)      >>= f = Input (λi → (h i) >>= f)
  (Output o r)   >>= f = Output o (r >>= f)
  (Return a)     >>= f = f a
```

We define the basic components: *input*: a pipe returning its received input, *output*: a pipe outputting a given o and returning (), and *return*: a pipe returning a given a.

```
input :: Pipe i o i
input = Input (λi → Return i)

output :: o → Pipe i o ()
output o = Output o (Return ())

return :: a → Pipe i o a
return a = Return a
```

The bind operation assembles these components into larger pipes. For example, *doubler*, a pipe which repeatedly takes its input, multiplies it by two and continually outputs this new value.

```
doubler :: Pipe Int Int a
doubler = do
  i ← input
  output (i ∗ 2)
  doubler
```

Another essential way of combining pipes is *connecting* them. This connects the outputs of the upstream to the inputs of the downstream. In the implementation, $connect_L$ performs a case analysis on the downstream q: if it is trying to output, we keep this intact, and we search for an input node. Once we find an input, then we call $connect_R$ on the wrapped continuation h and the upstream p. Then, in $connect_R$ we similarly scan the upstream for an output operation, keeping any input operations. If an output operation is found, the output value is passed to the continuation h and the connecting process starts again by calling $connect_L$. If at any point we see a return, then the connection finishes with this resulting return value. The implementation is given below.

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

```
connect :: Pipe i m a → Pipe m o a → Pipe i o a
connect p q = connect_L q p where
  connect_L :: Pipe m o a → Pipe i m a → Pipe i o a
  connect_L (Input h)      p = connect_R p h
  connect_L (Output o r) p = Output o (connect_L r p)
  connect_L (Return a)    p = Return a

  connect_R :: Pipe i m a → (m → Pipe m o a) → Pipe i o a
  connect_R (Input f)      h = Input (λv → connect_R (f v) h)
  connect_R (Output o r) h = connect_L (h o) r
  connect_R (Return a)    h = Return a
```

The connect operator enables expressing the connection of *doubler* with itself. In this example, the left *doubler* is the upstream and the right *doubler* is the downstream. The result of this connect is a pipe which outputs the quadruple of its incoming values.

```
quadrupler :: Pipe Int Int a
quadrupler = doubler `connect` doubler
```

We can run a pipe by interpreting it to *IO*.

```
toIO :: (Read i, Show o) ⇒ Pipe i o a → IO a
toIO (Input f) = do
  i ← readLn
  toIO (f i)
toIO (Output o r) = do
  putStrLn ("out: " ++ show o)
  toIO r
toIO (Return a) = return a
```

An example where we input *10*, receive *40* after passing the value through *quadrupler*, and then exit, is shown below.

```
λ> toIO quadrupler
10 ⟨Return⟩
out : 40
⟨Ctrl+C⟩
```

2.2 Three-continuation approach

The function *connect* is suboptimal because it has to recursively scan a pipe for an operation of interest while copying the other operation. When several connects are piled up, this leads to repeated scanning and copying of the same operations.

This section covers the *ContPipe* representation by Spivey, a different pipe representation which enables a faster connect implementation (Spivey, 2017). The code defining the representation and its connect function have been reproduced below. It features three

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

continuations, one for each constructor. The first continuation ($a \to Result\ i\ o$) represents the return constructor, this is called when the pipe terminates and returns the value a.

```
newtype ContPipe i o a = MakePipe {runPipe :: (a → Result i o) → Result i o}
type Result i o = InCont i → OutCont o → IO ()
```

The continuation $InCont\ i$ is the input continuation, it is resumed when the pipe wants to receive a value.

```
newtype InCont i = MakeInCont {resume_I :: OutCont i → IO ()}
```

The continuation $OutCont\ o$ is the output continuation, it is resumed when the pipe wants to output a value.

```
newtype OutCont o = MakeOutCont {resume_O :: o → InCont o → IO ()}
```

The monad instance is very similar to that of the continuation monad, in fact in Section 4.3 we deconstruct this type into a transformer stack with an outer layer of *ContT*.

```
instance Monad (ContPipe i o) where
  return a = MakePipe (λk → k a)
  p >>= f  = MakePipe (λk → runPipe p (λx → runPipe (f x) k))
```

In the following definitions for the basic pipe components, the continuation k is the return constructor—we give it a value and the input and output constructors and receive a pipe. The continuations k_i and k_o are the input and output constructors, we resume them with the newtype unwrapper and the continuations are refreshed once they have been used.

```
return :: a → ContPipe i o a
return a = MakePipe (λk k_i k_o → k a k_i k_o)
input :: ContPipe i o i
input = MakePipe (λk k_i k_o → resume_I k_i (MakeOutCont (λi k'_i → k i k'_i k_o)))
output :: o → ContPipe i o ()
output o = MakePipe (λk k_i k_o → resume_O k_o o (MakeInCont (λk'_o → k () k_i k'_o)))
```

Note that the names *return*, *input*, and *output* were already defined previously. We will reuse names for functions which are conceptually the same, but defined on different representations. We will add a subscript if it is needed for disambiguation.

We can use the *Monad* instance for *ContPipe* to compose pipes with **do**-notation, similar to *Pipe*.

```
doubler :: ContPipe Int Int a
doubler = do
  i ← input
  output (i * 2)
  doubler
```

We can also interpret *ContPipe* to *IO*.

```
toIO :: (Read i, Show o) ⇒ ContPipe i o () → IO ()
toIO p = runPipe p (λ() _ _ → return ()) k_i k_o where
```

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

```
ki = MakeInCont $ λko → do
        x ← readLn
        resumeO ko x ki
ko = MakeOutCont $ λo ki → do
        putStrLn ("out: " ++ show o)
        resumeI ki ko
```

The connect function for *ContPipe* is defined as:

```
connect :: ContPipe i m a → ContPipe m o a → ContPipe i o a
connect p q = MakePipe (λk ki ko →
  runPipe q err (MakeInCont (λk'o → runPipe p err ki k'o)) ko)
  where err = error "terminated"
```

With the connect definition, we are able to create the quadrupler pipe as before. Running *toIO quadrupler* results in an identical scenario to the *Pipe* scenario from the previous section.

```
quadrupler :: ContPipe Int Int a
quadrupler = doubler `connect` doubler
```

While Spivey has demonstrated the remarkable performance advantage of this connect operator, he sheds little light on the origin or underlying principles of the related encoding. The remainder of this paper provides this missing insight by deriving Spivey's efficient *ContPipe* representation from the ADT-style *Pipe* by means of well-known principles. The aim is to improve understanding of the applicability and limitations of the techniques used.

2.3 Overview

In the following sections, we derive this three-continuation encoding in a series of steps:

- First, we consider one-sided pipes which can either only produce or consume data. We derive the *InCont* and *OutCont* types from the definition of the *connect* function by eliminating its term-level mutual recursion.
- We rewrite the *connect* function in terms of *fold*. Each of the folds transforms the *Producer* and *Consumer* to the *InCont* and *OutCont* representation. Then, by using fold/build fusion, we eliminate the transformation step.
- Next, we apply the same principle to two-sided pipes as seen in this section, which results in the generalized representation of *Result*.
- As a final step to derive *ContPipe*, we wrap it with the continuation monad. Following the derivation in this way gives the three-continuation encoding over a generic monad m. By specializing m to *IO*, we obtain the type presented by Spivey.

3 Fast connecting for one-sided pipes

This section considers a simplified setting where pipes are one-sided, either only producing or only consuming data. For example, the *doubler* component cannot be defined in

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

this setting. Although this setting is not realistic in a practical sense, it is an instructional settings which only contains the core elements for our derivation. This simplified setting gives a more straightforward path to the fast connecting approach, which we generalize back to regular “mixed” pipes in Section 4.

3.1 One-sided pipes

In the simplified setting, pipes are either pure *Producer*s or pure *Consumer*s. A *Producer* *only* outputs values, while a *Consumer* *only* receives them.

```
data Producer o = Output o (Producer o)
data Consumer i = Input (i → Consumer i)
```

Now, *connect* only needs to consider the cases for *Input* on the left and *Output* on the right. The definition is

```
connect :: Producer b → Consumer b → a
connect p q = connect_L q p where
  connect_L :: Consumer b → Producer b → a
  connect_L (Input h) p = connect_R p h
  connect_R :: Producer b → (b → Consumer b) → a
  connect_R (Output o r) h = connect_L (h o) r
```

Example 3.1. The *Producer* data type is essentially an infinite stream of *o* values. A simple example is a stream of numbers increased by 1 each step.

```
prodFrom :: Int → Producer Int
prodFrom n = Output n (prodFrom (n + 1))
```

The *Consumer* data type processes elements of type *i*. Consumers at the end of a pipeline usually do something effectful with these elements. But, since our simplified consumer does not allow effectful operations yet, we utilize the impure *trace* function from the `Debug.Trace` module to inspect what the consumer is doing.

```
consumeDebug :: Show a ⇒ Consumer a
consumeDebug = Input (λa → trace ("CONSUMED: " ++ show a) consumeDebug)
```

We can connect *prodFrom* and *consumeDebug* using *connect*, this results in the *prodFrom* producer feeding each of its values to the *consumeDebug* consumer. When we try to evaluate the value of the connection, we see a debug line for each of the values created by the producer. The result of the connected result is never seen since it continues indefinitely; thus, we have to forcefully stop the execution of this connect.

```
λ > prodFrom 1 `connect` consumeDebug
CONSUMED : 1
CONSUMED : 2
CONSUMED : 3
CONSUMED : 4
⟨Ctrl+C⟩
```

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

Note that these definitions are not very useful from a user perspective: without impure IO it is impossible to obtain any result from connecting a producer and consumer. However, we chose these definitions as they capture the essence of the topic in the next section.

3.2 Mutual recursion elimination

This section contains the key insight of the approach: the three-continuation representation can be derived from eliminating the mutual recursion in the *connect* function.

The two auxiliary functions $connect_L$ and $connect_R$ turn, respectively, a producer and a consumer into the result of type a by means of an additional parameter, which is respectively of type (*Producer* b) and ($b \to$ *Consumer* b). To highlight these parameters, we introduce type synonyms for them.

```
type ProdPar' b = Producer b
type ConsPar' b = b → Consumer b
```

Now we refactor $connect_L$ and $connect_R$ with respect to their additional parameter in a way that removes the term-level mutual recursion between them. Consider $connect_L$ which does not use its parameter p directly, but only its interpretation by function $connect_R$. We refactor this code to a form where $connect_R$ has already been applied to p before it is passed to $connect_L$. This adapted $connect_L$ would then have type *Consumer* $b \to$ (*ConsPar'* $b \to a) \to a$. At the same time, we apply a similar transformation to $connect_R$, moving the application of $connect_L$ to h out of it. This yields infinite types for the two new parameters, which Haskell only accepts if we wrap them in newtypes.

```
newtype ProdPar b a = ProdPar (ConsPar b a → a)
newtype ConsPar b a = ConsPar (b → ProdPar b a → a)
```

The connect function is then defined by appropriately placing newtype (un-)wrappers.

```
connect :: Producer b → Consumer b → a
connect p q = ml q (ProdPar (mr p)) where
  ml :: Consumer b → ProdPar b a → a
  ml (Input h) (ProdPar p) = p (ConsPar (λi → (ml (h i))))
  mr :: Producer b → ConsPar b a → a
  mr (Output o r) (ConsPar h) = h o (ProdPar (mr r))
```

Note that we can recover Spivey's *InCont* i and *OutCont* o by instantiating the type parameter a to *IO* () in *ProdPar* $i\ a$ and *ConsPar* $o\ a$, respectively.

3.3 Structural recursion with fold

Due to the removal of the term-level mutual recursion in *ml* and *mr*, they are easily adapted to their structurally recursive form. By isolating the work done in each recursive step, we obtain alg_L and alg_R.

```
type Carrier_L i a = ProdPar i a → a
alg_L :: (i → Carrier_L i a) → Carrier_L i a
```

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

$$alg_L\, f = \lambda(ProdPar\; prod) \rightarrow prod\; (ConsPar\; f)$$

$$\textbf{type}\; Carrier_R\; o\; a = ConsPar\; o\; a \rightarrow a$$

$$alg_R :: o \rightarrow Carrier_R\; o\; a \rightarrow Carrier_R\; o\; a$$
$$alg_R\; o\; prod = \lambda(ConsPar\; cons) \rightarrow cons\; o\; (ProdPar\; prod)$$

The functions alg_L and alg_R are now in a form known as algebras. Algebras are a combination of a *carrier* r, the type of the resulting value, and an *action* of type $f\; r \rightarrow r$. This action denotes the computation performed at each node of the recursive data type, for which the functor f determines the shape of its nodes. We omit the carrier type if it is clear from the context and simply refer to an algebra by its action.

The structural recursion schemes, or *folds*, for *Consumer* and *Producer* take algebras of the form $(i \rightarrow r) \rightarrow r$ and $o \rightarrow r \rightarrow r$ and interpret the data types to a result r using these functions. Their definitions are

$$fold_P :: (o \rightarrow r \rightarrow r) \rightarrow Producer\; o \rightarrow r$$
$$fold_P\; alg\; (Output\; o\; r) = alg\; o\; (fold_P\; alg\; r)$$

$$fold_C :: ((i \rightarrow r) \rightarrow r) \rightarrow Consumer\; i \rightarrow r$$
$$fold_C\; alg\; (Input\; h) = alg\; (\lambda i \rightarrow fold_C\; alg\; (h\; i))$$

Example 3.2. An example use of folds is an interpretation to *IO* by supplying the inputs for a consumer or printing the outputs of a producer.

```
type CarrierCons_IO i = IO ()
consume_IO :: Read i ⇒ Consumer i → IO ()
consume_IO = fold_C alg where
  alg :: Read i ⇒ (i → CarrierCons_IO i) → CarrierCons_IO i
  alg f = do
    x ← readLn
    f x
type CarrierProd_IO o = IO ()
produce_IO :: Show o ⇒ Producer o → IO ()
produce_IO = fold_P alg where
  alg :: Show o ⇒ o → CarrierProd_IO o → CarrierProd_IO o
  alg o p = do
    print o
    p
```

Another example is expressing *connect* with folds using alg_L and alg_R.

$$connect_{fold} :: Producer\; x \rightarrow Consumer\; x \rightarrow a$$
$$connect_{fold}\; p\; q = fold_C\; alg_L\; q\; (ProdPar\; (fold_P\; alg_R\; p))$$

3.4 A shortcut to a connect-friendly representation

Instead of directly defining a *Consumer* or *Producer* value in terms of the data constructors of the respective types, we can also do it in a more roundabout way by abstracting over the

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

constructor occurrences—this is known as *build* form (Gill *et al.*, 1993). The *build* function then instantiates the abstracted constructors with the actual constructors; for *Consumer* and *Producer* they are

$$build_C :: (\forall r.((i \to r) \to r) \to r) \to Consumer\ i$$
$$build_C\ g = g\ Input$$
$$build_P :: (\forall r.(o \to r \to r) \to r) \to Producer\ o$$
$$build_P\ g = g\ Output$$

For instance, *prodFrom*, which was defined in an earlier example:

$$prodFrom :: Int \to Producer\ Int$$
$$prodFrom\ n = Producer\ n\ (prodFrom\ (n + 1))$$

can be written using its *build* form as:

$$buildFrom :: Int \to Producer\ Int$$
$$buildFrom\ n = build_P\ (buildHelper\ n)$$
$$buildHelper :: Int \to (\forall r.(Int \to r \to r) \to r)$$
$$buildHelper\ n\ p = go\ n\ \mathbf{where}$$
$$\quad go\ n = p\ n\ (go\ (n + 1))$$

The motivation for these build functions is to optimize using the fold/build fusion rule, a special form of shortcut fusion (Ghani *et al.*, 2004). This rule can be applied when a fold directly follows a build, specifically for *Consumer* and *Producer* these fusion rules are

$$fold_C\ alg\ (build_C\ cons) = cons\ alg$$
$$fold_P\ alg\ (build_P\ prod) = prod\ alg$$

In other words, instead of first building an ADT representation and then folding it to its result, we can directly create the result of the fold. This readily applies to the two folds in $connect_{fold}$. We can directly represent consumers and producers in terms of the carrier types of those two folds,

$$\mathbf{type}\ Consumer_{Alt}\ i = \forall a.Carrier_L\ i\ a \quad \text{-- } \forall a.ProdPar\ i\ a \to a$$
$$\mathbf{type}\ Producer_{Alt}\ o = \forall a.Carrier_R\ o\ a \quad \text{-- } \forall a.ConsPar\ o\ a \to a$$

using their algebras as constructors:

$$input :: (i \to Consumer_{Alt}\ i) \to Consumer_{Alt}\ i$$
$$input = alg_L$$
$$output :: o \to Producer_{Alt}\ o \to Producer_{Alt}\ o$$
$$output = alg_R$$

For example, if we fold *buildFrom* using alg_R we get this:

$$buildFrom :: Int \to Producer_{Alt}\ Int$$
$$buildFrom\ n = fold_P\ alg_R\ (build_P\ (buildHelper\ n))$$

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

This matches the condition for the fold/build fusion rule, so this is equivalent to the following definition:

$$buildFrom :: Int \to Producer_{Alt}\ Int$$
$$buildFrom\ n = buildHelper\ n\ alg_R$$

After simplifying and using the *output* constructor instead of alg_R, we get the following definition:

$$buildFrom :: Int \to Producer_{Alt}\ Int$$
$$buildFrom\ n = output\ n\ (buildFrom\ (n + 1))$$

This definition is very similar to the *prodFrom* definition from Example 3.1, but we use the *output* constructor instead of the *Producer* constructor.

We can follow a similar pattern for the $connect_{fold}$ function. By expressing p and q in their *build* forms.

$$connect :: (\forall r.(b \to r \to r) \to r) \to (\forall r.((b \to r) \to r) \to r) \to a$$
$$connect\ p\ q = fold_C\ alg_L\ (build_C\ q)\ (ProdPar\ (fold_P\ alg_R\ (build_P\ p)))$$

Using the fold/build fusion rule, the *fold* and *build* functions disappear.

$$connect :: (\forall r.(b \to r \to r) \to r) \to (\forall r.((b \to r) \to r) \to r) \to a$$
$$connect\ p\ q = q\ alg_L\ (ProdPar\ (p\ alg_R))$$

If we assume that we write our producers and consumers using the constructors for $Producer_{Alt}$ and $Consumer_{Alt}$ directly, then the connect function for this representation is an almost trivial operation.

$$connect :: Producer_{Alt}\ b \to Consumer_{Alt}\ b \to a$$
$$connect\ p\ q = q\ (ProdPar\ p)$$

Example 3.3. We redo Example 3.1, but now utilizing the alternative constructors and connect function.

For our producer, which generates increasing numbers, we utilize *buildFrom* which we created in the previous section. Then we create the debugging consumer by replacing the *Consumer* constructor in *consumeDebug* from Example 3.1 with *input*.

$$consumeDebug :: Show\ a \Rightarrow Consumer_{Alt}\ a$$
$$consumeDebug = input\ (\lambda a \to trace\ (\texttt{"CONSUMED: "} +\!\!+\ show\ a)\ consumeDebug)$$

Evaluating the connection of this producer and consumer results in exactly the same scenario as in the previous example.

λ > *buildFrom* 1 ‘*connect*‘ *consumeDebug*
CONSUMED : 1
CONSUMED : 2
CONSUMED : 3
CONSUMED : 4
⟨Ctrl+C⟩

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

3.5 A not so special representation

These connect-friendly representations of producers and consumers are not just specializations; they are in fact isomorphic to the originals. The inverses of ml and mr to complete the isomorphism are given by ml^{-1} and mr^{-1}. The proof can be found in Appendix A.

$$
\begin{aligned}
&ml^{-1} :: Consumer_{Alt}\ i \to Consumer\ i \\
&ml^{-1}\ f = f\ (ProdPar\ h)\ \textbf{where} \\
&\quad h :: ConsPar\ i\ (Consumer\ i) \to Consumer\ i \\
&\quad h\ (ConsPar\ f) = Input\ (\lambda x \to f\ x\ (ProdPar\ h)) \\
&mr^{-1} :: Producer_{Alt}\ o \to Producer\ o \\
&mr^{-1}\ f = f\ (ConsPar\ (\lambda x\ p \to Output\ x\ (h\ p)))\ \textbf{where} \\
&\quad h :: ProdPar\ o\ (Producer\ o) \to Producer\ o \\
&\quad h\ (ProdPar\ f) = f\ (ConsPar\ (\lambda x\ p \to Output\ x\ (h\ p)))
\end{aligned}
$$

Hence, we can also fold with other algebras by transforming the connect-friendly representation back to the ADT, and then folding over that.

$$
\begin{aligned}
&fold_{P_{Alt}} :: (o \to a \to a) \to Producer_{Alt}\ o \to a \\
&fold_{P_{Alt}}\ alg\ rep = fold_P\ alg\ (mr^{-1}\ rep) \\
&fold_{C_{Alt}} :: ((i \to a) \to a) \to Consumer_{Alt}\ i \to a \\
&fold_{C_{Alt}}\ alg\ rep = fold_C\ alg\ (ml^{-1}\ rep)
\end{aligned}
$$

Of course, these definitions are wasteful because they create intermediate data types. However, by performing fold/build fusion, we obtain their fused versions:

$$
\begin{aligned}
&fold_{P_{Alt}}\ alg\ rep = rep\ (ConsPar\ (\lambda x\ p \to alg\ x\ (h\ p)))\ \textbf{where} \\
&\quad h\ (ProdPar\ f) = f\ (ConsPar\ (\lambda x\ p \to alg\ x\ (h\ p))) \\
&fold_{C_{Alt}}\ alg\ rep = rep\ (ProdPar\ h)\ \textbf{where} \\
&\quad h\ (ConsPar\ f) = alg\ (\lambda x \to f\ x\ (ProdPar\ h))
\end{aligned}
$$

4 Return to two-sided pipes

The previous section has derived an alternative approach for connecting simplified *Consumer* and *Producer* pipes. This section extends that approach to proper *Pipe*s in two steps, first supporting both input and output operations, and then also a *return*.

4.1 Pipe of no return

Let us consider pipes with both input and output operations, but no *return*. Since these pipes are infinite because they never return a value, we give them the subscript $_\infty$.

$$
\begin{aligned}
\textbf{data}\ Pipe_\infty\ i\ o &= Input_\infty\ (i \to Pipe_\infty\ i\ o) \\
&\mid Output_\infty\ o\ (Pipe_\infty\ i\ o)
\end{aligned}
$$

We can fold over these pipes by providing algebras for both the input and output operation, agreeing on the carrier type a.

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

```
foldPipe∞ :: Pipe∞ i o → ((i → a) → a) → (o → a → a) → a
foldPipe∞ p inAlg outAlg = go p where
  go (Input∞ p)    = inAlg (λi → go (p i))
  go (Output∞ o p) = outAlg o (go p)
```

To connect these pipes, we use alg_L and alg_R developed in the previous section. There is only one snag: the two algebras do not agree on the carrier type. The carrier types were the alternate representations $Consumer_{Alt}$ and $Producer_{Alt}$.

```
type ConsumerAlt i = ∀a.ProdPar i a → a
type ProducerAlt o = ∀a.ConsPar o a → a
```

We reconcile these two carrier types by observing that both are functions with a common result type, but different parameter types. A combination of both is a function taking both parameter types as input.

```
type ResultR i o = ∀a.ConsPar o a → ProdPar i a → a
```

The algebra actions are easily adapted to the additional parameter. They simply pass it on to the recursive positions without using it themselves.

```
input :: (i → ResultR i o) → ResultR i o
input f = λcons (ProdPar prod) →
  prod (ConsPar (λi prod′ → f i cons prod′))
output :: o → ResultR i o → ResultR i o
output o result = λ(ConsPar cons) prod →
  cons o (ProdPar (λcons′ → result cons′ prod))
```

Like before, we can avoid the ADT $Pipe_\infty$ and directly work with $Result_R$ using the algebras as constructor functions.

Finally, we can use the one-sided connect function from the previous section to connect the output side of a $Result_R\ i\ m$ pipe with the input side of a $Result_R\ m\ o$ pipe. Because we defer the interpretation of the i and o sides of the respective pipes, this one-sided connect does not yield a result of type a, but rather one of type $ConsPar\ o\ a \to ProdPar\ i\ a \to a$. In other words, the connection of the two pipes yields a $Result_R\ i\ o$ pipe.

```
connect :: ResultR i m → ResultR m o → ResultR i o
connect p q = λconso prodi →
  let q′ = q conso
      p′ = flip p prodi
  in q′ (ProdPar p′)
```

Example 4.1. We redefine the producer and consumer from Example 3.1 using the $Result_R$ type and its constructors.

```
prodFrom :: Int → ResultR i Int
prodFrom n = output n (prodFrom (n + 1))
consumeDebug :: Show a ⇒ ResultR a o
consumeDebug =
  input (λa → trace ("CONSUMED: " ++ show a) consumeDebug)
```

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

The result of connecting this producer and consumer gives us back a $Result_R\ i\ o$. This is a function $\forall a. ConsPar\ o\ a \rightarrow ProdPar\ i\ a \rightarrow a$, where $ConsPar\ o\ a$ is the interpretation of the outer outputs and $ProdPar\ i\ a$ is the interpretation of the outer inputs. If there are no outer inputs or outputs, we can pass dummy implementations by instantiating the type variables to (). This is defined in the function *runEffect*.

```
runEffect :: Result_R () () → a
runEffect p = p consPar prodPar where
  consPar = ConsPar (λv (ProdPar f) → f consPar)
  prodPar = ProdPar (λ(ConsPar f) → f () prodPar)
```

This results in the same scenario as in the previous examples:

```
λ> runEffect $ prodFrom 1 `connect` consumeDebug
CONSUMED : 1
CONSUMED : 2
CONSUMED : 3
CONSUMED : 4
⟨Ctrl+C⟩
```

Using the capabilities of $Result_R$ to do both input and output, we can again define the doubling component. This component continually multiplies each of its incoming values by two and then outputs it again.

```
doubler :: Result_R Int Int
doubler =
  input $ λi →
  output (i ∗ 2) $
  doubler
```

Which results in the following output once we place the doubler component in the middle of our previous pipeline.

```
λ> runEffect $
  prodFrom 1 `connect` doubler `connect` consumeDebug
CONSUMED : 2
CONSUMED : 4
CONSUMED : 6
CONSUMED : 8
⟨Ctrl+C⟩
```

4.2 Return to *return*

Finally, we reobtain *return* and the monadic structure of pipes in a slightly unusual way, by means of the *continuation* monad.

```
newtype Cont r a = Cont {runCont :: (a → r) → r}
instance Monad (Cont r) where
```

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

$$\begin{aligned} &\mathit{return}\ x = \mathit{Cont}\ (\lambda k \to k\ x) \\ &p \gg\!= f\ \ = \mathit{Cont}\ (\lambda k \to \mathit{runCont}\ p\ (\lambda x \to \mathit{runCont}\ (f\ x)\ k)) \end{aligned}$$

If we specialize the result type r to $\mathit{Result}_R\ i\ o$, we get

$$\mathbf{newtype}\ \mathit{ContP}\ i\ o\ a = \mathit{ContP}\ ((a \to \mathit{Result}_R\ i\ o) \to \mathit{Result}_R\ i\ o)$$

The connect function for *ContP* is implemented in terms of *connect*.

$$\begin{aligned} &\mathit{connect} :: \mathit{ContP}\ i\ m\ \mathit{Void} \to \mathit{ContP}\ m\ o\ \mathit{Void} \to \mathit{ContP}\ i\ o\ a \\ &\mathit{connect}\ (\mathit{ContP}\ p)\ (\mathit{ContP}\ q) = \mathit{ContP}\ (\lambda k \to \mathit{connect}\ (p\ \mathit{absurd})\ (q\ \mathit{absurd})) \end{aligned}$$

However, there is an issue: before *connect* can connect the two pipes, their continuations (the interpretations of the *return* constructor) must be supplied. Yet, the resulting pipe's continuation type k does not match that of either the upstream or downstream pipe. Thus, it seems that we are stuck, unless we assume what we have been all along: that the two pipes are infinite. Indeed, in that case it does not matter that we do not have a continuation for them, as their continuation is never reached anyway. In short, *connect* only works for never-returning pipes, which we signal with the return type *Void*, only inhabited by $\bot$.

4.3 Specialization for *IO*

To get exactly Spivey's representation, we instantiate the polymorphic type variable a in $\mathit{Result}_R\ i\ o$ to $\mathit{IO}\ ()$, which yields:

$$\begin{aligned} &\mathbf{type}\ \mathit{Result}\ i\ o = \mathit{InCont}\ i \to \mathit{OutCont}\ o \to \mathit{IO}\ () \\ &\mathbf{newtype}\ \mathit{ContPipe}\ i\ o\ a = \mathit{MakePipe}\ \{\mathit{runPipe} :: (a \to \mathit{Result}\ i\ o) \to \mathit{Result}\ i\ o\} \end{aligned}$$

The *Monad* instance for the *ContPipe* representation can be constructed by viewing it as a type created from a monad transformer stack. The stack consists of a *ContT* layer, two *ReaderT* layers, and an inner *IO* layer. Using the `DerivingVia` Glasgow Haskell Compiler (GHC) extension (Blöndal *et al.*, 2018), the *Monad* instance can be derived using this alternate representation.

Each *ReaderT* layer contains respectively an input and output continuation, which results in the following alternate formulation of *Result*.

$$\begin{aligned} &\mathbf{newtype}\ \mathit{ReaderT}\ r\ m\ a = \mathit{ReaderT}\ \{\mathit{runReaderT} :: r \to m\ a\} \\ &\mathbf{type}\ \mathit{Result}'\ i\ o = \mathit{ReaderT}\ (\mathit{InCont}\ i)\ (\mathit{ReaderT}\ (\mathit{OutCont}\ o)\ \mathit{IO})\ () \end{aligned}$$

Then, to obtain *ContPipe* we add the *ContT* layer, since $\mathit{Cont}\ (m\ r)$ is equal to $\mathit{ContT}\ r\ m$ for any monad m. This results in the following alternate formulation of *ContPipe*:

$$\begin{aligned} &\mathbf{newtype}\ \mathit{ContT}\ r\ m\ a = \mathit{ContT}\ \{\mathit{runContT} :: (a \to m\ r) \to m\ r\} \\ &\mathbf{type}\ \mathit{ContPipe}'\ i\ o\ a = \mathit{ContT}\ ()\ (\mathit{ReaderT}\ (\mathit{InCont}\ i)\ (\mathit{ReaderT}\ (\mathit{OutCont}\ o)\ \mathit{IO}))\ a \end{aligned}$$

The *exit* and *effect* operations presented by Spivey are specializations of the *abort* and *lift* operations available to the *ContT* monad transformer.

```
  -- ContT generic operations
abort :: m r → ContT r m a
abort r = ContT (λk → r)
```

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

$lift_{ContT} :: Monad\ m \Rightarrow m\ a \to ContT\ r\ m\ a$
$lift_{ContT}\ p = ContT\ (\lambda k \to p \gg= k)$

-- ContPipe specific operations

$exit :: ContPipe'\ i\ o\ a$
$exit = abort\ (lift_{ReaderT}\ (lift_{ReaderT}\ (return\ ())))$
$effect :: IO\ a \to ContPipe'\ i\ o\ a$
$effect\ e = lift_{ContT}\ (lift_{ReaderT}\ (lift_{ReaderT}\ e))$

We obtain the constructor and the connect functions by adapting the functions of the three-continuation encoding, shown in Section 2.2, to fit the additions of *ContT* and *ReaderT*.

$input :: ContPipe'\ i\ o\ i$
$input = ContT\ (\lambda k \to ReaderT\ (\lambda k_i \to ReaderT\ (\lambda k_o \to$
$\quad resume_I\ k_i\ (MakeOutCont\ (\lambda i\ k_i' \to runReaderT\ (runReaderT\ (k\ i)\ k_i')\ k_o)))))$
$output :: o \to ContPipe'\ i\ o\ ()$
$output\ o = ContT\ (\lambda k \to ReaderT\ (\lambda k_i \to ReaderT\ (\lambda k_o \to$
$\quad resume_O\ k_o\ o\ (MakeInCont\ (\lambda k_o' \to runReaderT\ (runReaderT\ (k\ ())\ k_i)\ k_o')))))$

$connect :: ContPipe'\ i\ m\ a \to ContPipe'\ m\ o\ a \to ContPipe'\ i\ o\ a$
$connect\ p\ q = ContT\ (\lambda k \to ReaderT\ (\lambda k_i \to ReaderT\ (\lambda k_o \to$
$\quad runReaderT\ (runReaderT\ (runContT\ q\ err)$
$\qquad (MakeInCont\ (\lambda k_o' \to runReaderT\ (runReaderT\ (runContT\ p\ err)\ k_i)\ k_o')))$
$\qquad k_o)))$
$\quad$ **where** $err = error$ "terminated"

5 Bidirectional pipes

So far we have covered unidirectional pipes where information flows in one direction through the pipe, from the *output* operations in one pipe to the *input* operations in the next pipe downstream. However, some use cases also require information to flow upstream and pipes that support this are called bidirectional.

Bidirectional pipes have use cases in a variety of use cases: piggybacking extra information into upstream *inputs* (e.g., when the number of requested bytes needs to be specified), automatic tracking of leftover input to upstream requests in effectful streams, closing the upstream end (by setting the request type to *Void*), or for implementing a structure reminiscent of a reverse proxy.

First we give an introduction to the bidirectional pipes setting and an example of the reverse proxy use case in Section 5.1. Then, we derive the three-continuation encoding for the bidirectional setting by applying the methods from the previous sections. We obtain the resulting encoding without any additional complications.

5.1 ADT encoding

The *Proxy* data type at the core of the `pipes` library (Gonzalez, 2012) implements bidirectional pipes. The operations *request* and *respond* are respectively downstream and

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

upstream combinations of *input* and *output*. We can embed effects of the monad *m* with the constructor *M*.

```
data Proxy a' a b' b m r = Request a' (a → Proxy a' a b' b m r)
                         | Respond b (b' → Proxy a' a b' b m r)
                         | M         (m (Proxy a' a b' b m r))
                         | Pure r
```

The *Proxy* data type is a monad, similarly to *Pipe*.

```
instance Functor m ⇒ Monad (Proxy a' a b' b m) where
  return = Pure
  p0 >>= f = go p0 where
    go p = case p of
      Request a' fa → Request a' (λa → go (fa a))
      Respond b fb' → Respond b (λb' → go (fb' b'))
      M         m → M (go <$> m)
      Pure r      → f r
```

In addition, it is a monad transformer due to the *M* constructor.

```
instance MonadTrans (Proxy a' a b' b) where
  lift m = M (Pure <$> m)
```

We use the following two helper constructors for the *request* and *respond* operations.

```
request :: a' → Proxy a' r b' b m r
request x = Request x Pure

respond :: b → Proxy a' a r b m r
respond x = Respond x Pure
```

The functions +>> and >>~ correspond to the functions $connect_L$ and $connect_R$ extended for the bidirectional pipes. The functions follow the same idea as the original ADT connect function, but take into consideration the extra parameters for the request/respond operations. Below is the implementation as seen in the `pipes` source code.

```
(+>>) :: Functor m
      ⇒ (b' → Proxy a' a b' b m r) → Proxy b' b c' c m r
      → Proxy a' a c' c m r
fb' +>> p = case p of
  Request b' fb  → fb' b' >>~ fb
  Respond c  fc' → Respond c (λc' → fb' +>> fc' c')
  M          m   → M ((λp' → fb' +>> p') <$> m)
  Pure       r   → Pure r

(>>~) :: Functor m
      ⇒ Proxy a' a b' b m r → (b → Proxy b' b c' c m r)
      → Proxy a' a c' c m r
p >>~ fb = case p of
  Request  a' fa → Request a' (λa → fa a >>~ fb)
```

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

```
  Respond b fb' → fb' +>> fb b
  M         m   → M ((λp' → p' >>~ fb) <$> m)
  Pure      r   → Pure r
```

Example 5.1. First, let us take a look at a simple use case for bidirectional pipes: interactions between a client and server. We implement a simplified version of Hoogle: a server which holds a mapping from function names to function type signatures. The client can request a function name and the server replies with the corresponding signature.

The client implementation requests input from the user, and then does a *request* for the type signature. This *request* also awaits the reply of the server, which the client prints on the screen once it is received. This process is then looped forever.

```
client :: Proxy String String x' x IO Void
client = forever $ do
  lift (putStrLn "Hoogle Search")
  x ← lift getLine
  resp ← request x
  lift (print resp)
```

The server implementation *respond*s to any incoming requests for type signatures. The *respond* function awaits the request of the client, but also needs as input the response data to the client. The server is constructed as a function taking the client request as input, and once it is known a response is sent. This response immediately awaits a new request for the server and the function calls itself with this new client request as input.

```
server :: (Functor m) ⇒ String → Proxy x' x String String m Void
server clientRequest = do
  next ← respond (hoogleDB clientRequest)
  server next
  where
    hoogleDB "map" = "(a -> b) -> [a] -> [b]"
    hoogleDB "filter" = "(a -> Bool) -> [a] -> [a]"
    hoogleDB "foldr" = "(a -> b -> b) -> b -> [a] -> b"
    hoogleDB x = "error: " ++ x
```

Then, we can connect the bidirectional pipes *client* and *server* with $+\gg$.

```
app :: Proxy a' a b' b IO Void
app = server +>> client
```

This gives a *Proxy* a' a b' b *IO Void*, which is a *Proxy* with no lingering incoming or outgoing connections. We can run this using the *runEffect* function which runs all the effects in the *Proxy*.

```
runEffect :: Monad m ⇒ Proxy Void () () Void m r → m r
runEffect = go where
  go p = case p of
    Request v _ → absurd v
```

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

```
    Respond v _ → absurd v
    M m    → m >>= go
    Pure r → return r
```

Below is an example where we run *app* and pass it the inputs `"map,"` `"filter,"` and `"mop"` and then quit the session.

```
λ > runEffect app
Hoogle Search
map
"(a -> b) -> [a] -> [b]"
Hoogle Search
filter
"(a -> Bool) -> [a] -> [a]"
Hoogle Search
mop
"unknown function: mop"
Hoogle Search
⟨Ctrl+C⟩
```

Now we extend this system by putting an intermediate component, the mediator, which both *request*s and *respond*s. This mediator tries to fix misspellings of function names by the client and prettify the server output.

```
mediator :: (Functor m) ⇒ String → Proxy String String String String m Void
mediator originalClientRequest = do
  let (fixClientRequest, fixed) = case correct originalClientRequest of
    (Just fix) → (fix, True)
    Nothing → (originalClientRequest, False)
  serverResponse ← request (fixClientRequest)
  let serverErr = "error" `isPrefixOf` serverResponse
  next ← respond $ case (serverErr, fixed) of
    (True, _) → ("Function " ++ fixClientRequest ++ " is unknown.")
    (False, False) →
      "Type for " ++ fixClientRequest ++ " is: " ++ serverResponse
    (False, True) → "Type for " ++ fixClientRequest ++ " is: " ++
      serverResponse ++ " (fixed from " ++ originalClientRequest ++ ")"
  mediator next
  where
    correct "mop" = Just "map"
    correct _ = Nothing
```

Now we can place the mediator component inbetween server and client. Below, the client requests `"map"` and `"mop"` and both requests succeed since the misspelling `"mop"` has been fixed by the mediator.

```
λ > runEffect (server +>> (mediator +>> client))
Hoogle Search
map
```

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

```
"Type for map is: (a -> b) -> [a] -> [b]"
Hoogle Search
mop
"Type for map is: (a -> b) -> [a] -> [b] (fixed from mop)"
Hoogle Search
⟨Ctrl+C⟩
```

5.2 Constructing the continuation-based encoding

In this section, we construct the continuation-based representation for bidirectional pipes by following the derivation of Sections 3 and 4.

5.2.1 One-sided bidirectional pipes

First, we take a look at the one-sided setting for bidirectional pipes. This means that we can only express clients and servers, but no mediators. Both a client and server can be expressed with the *ProdCons* data type, which is both a producer of o values and a consumer of i values.

$$\textbf{data}\ \mathit{ProdCons}\ i\ o = \mathit{ProdCons}\ o\ (i \to \mathit{ProdCons}\ i\ o)$$

The two one-sided bidirectional components are then: the client which only does requests, and the server which only does responses.

$$\textbf{type}\ \mathit{Client}\ b'\ b = \mathit{ProdCons}\ b'\ b$$
$$\textbf{type}\ \mathit{Server}\ b'\ b = \mathit{ProdCons}\ b\ b'$$

The connect function for this setting is a simplification of the $+\!\gg'$ and $\gg\!\sim'$ functions, or it can be seen as a generalization of $\mathit{connect}_L$ and $\mathit{connect}_R$.

$$(+\!\gg') :: \mathit{Server}\ i\ o \to (i \to \mathit{Client}\ i\ o) \to a$$
$$(+\!\gg')\ (\mathit{ProdCons}\ i\ \mathit{fo})\ \mathit{fi} = (\mathit{fi}\ i)\ +\!\gg'\ \mathit{fo}$$
$$(\gg\!\sim') :: \mathit{Client}\ i\ o \to (o \to \mathit{Server}\ i\ o) \to a$$
$$(\gg\!\sim')\ (\mathit{ProdCons}\ o\ \mathit{fi})\ \mathit{fo} = (\mathit{fo}\ o)\ \gg\!\sim'\ \mathit{fi}$$

After carefully looking at these functions, we can conclude that they are actually the same.

$$\mathit{connect} :: \mathit{ProdCons}\ i\ o \to (o \to \mathit{ProdCons}\ o\ i) \to a$$
$$\mathit{connect}\ (\mathit{ProdCons}\ o\ \mathit{fi})\ \mathit{fo} = \mathit{connect}\ (\mathit{fo}\ o)\ \mathit{fi}$$

Now we give a name to the parameter $o \to \mathit{ProdCons}\ o\ i$, namely $\mathit{PCPar}'\ i\ o$. Then, we replace this with $\mathit{PCPar}\ i\ o\ a$, the result after processing this parameter with connectPC within the $o \to -$ functor. This gives us $\mathit{PCPar}\ i\ o\ a = o \to (i \to \mathit{ProdCons}\ o\ i) \to a$, and we replace the $i \to \mathit{ProdCons}\ o\ i$ with $\mathit{PCPar}\ i\ o\ a$ since that is the assumption we started from. As a result, we obtain the *PCPar* representation and the new connect function.

$$\textbf{newtype}\ \mathit{PCPar}\ i\ o\ a = \mathit{PCPar}\ \{\mathit{unPCPar} :: o \to \mathit{PCPar}\ o\ i\ a \to a\}$$

$$\mathit{connect} :: \mathit{ProdCons}\ i\ o \to \mathit{PCPar}\ i\ o\ a \to a$$
$$\mathit{connect}\ (\mathit{ProdCons}\ o\ \mathit{fi})\ \mathit{fo} = \mathit{unPCPar}\ \mathit{fo}\ o\ (\mathit{PCPar}\ (\mathit{connect} \circ \mathit{fi}))$$

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

5.2.2 Two-sided bidirectional pipes

Now we can construct the combined representation for *Proxy* by combining the alternate form of the request and respond operation, the *PCPar* type. In addition, we add the $m\ r \to r$ parameter which is the church encoding of the lift operation.

```
type ProxyRep a' a b' b m = ∀r.
  PCPar a a' r →     -- request
  PCPar b' b r →     -- respond
  (m r → r) →        -- lift
  r
```

Connecting these representations proceeds in a similar manner as for $Result_R$. We pass the interpretations of the intermediate connections to their corresponding sides and defer the interpretation of the outer connections.

```
connect :: (c' → ProxyRep a' a c' c m) → ProxyRep c' c b' b m → ProxyRep a' a b' b m
connect fc' q = λreq res m →
  let p' c' res = fc' c' req res m
      q' req = q req res m
  in q' (PCPar p')
```

And as a last step, we obtain the monadic structure by wrapping the representation with the continuation monad. Again, the same issue concerning the *return* constructor arises.

```
newtype ContPr a' a b' b m r = ContPr {unContPr ::
  (r → ProxyRep a' a b' b m) → ProxyRep a' a b' b m}
connect ::
  (c' → ContPr a' a c' c m Void) → ContPr c' c b' b m Void → ContPr a' a b' b m r
connect fc' (ContPr q) = ContPr (λk →
  connect (λc' → unContPr (fc' c') absurd) (q absurd))
```

6 Benchmarks

This section discusses various benchmarks comparing this alternate representation with three stream processing frameworks in the Haskell ecosystem: `pipes`, `conduit`, and `streamly`. All benchmarks are executed using the `criterion` library (O'Sullivan, 2009) on an Intel Core i7-6600U at 2.60 GHz with 8 GB memory running Ubuntu 16.04 and GHC 8.8.3, with `-O2` enabled. In the following sections, we highlight the relevant parts of the benchmarking code, but we refer to the full implementation for the details (Pieters, 2018a).

The library versions used are `pipes` v4.3.13, `conduit` v1.3.2, and `streamly` v0.7.1. A short discussion on `conduit` and `streamly` can be found in Section 7. These are compared to our generalized form (`proxyrep`) of the continuation-based representation. We include Spivey's original code (`contpipe`) where possible to ensure that our generalizations have retained the performance characteristics.

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

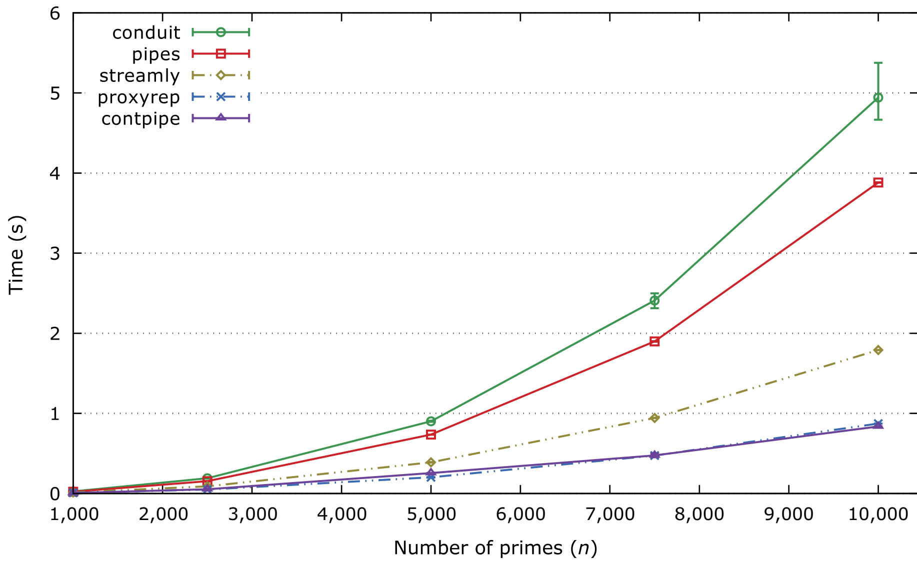

Fig. 1. Results of the `primes` benchmark.

6.1 Spivey's benchmarks

We have replicated Spivey's benchmarks `primes`, `deep-par`, and `deep-seq`. The results of replicating these benchmarks are what we expect, given the findings of Spivey.

6.1.1 `primes` benchmark

Figure 1 shows the results of the `primes` benchmark, which calculates the first n primes by accumulating filter operations using the *connect* operation. The x-axis shows the number of primes which are generated and the y-axis shows the runtime (lower is better).

The essence of this benchmark is the following code:

```
sieve :: Pipe Int Int x
sieve = do
  p ← await
  yield p
  filter (λx → x ‘mod‘ p ≢ 0) ‘connect‘ sieve
primes :: Int → Pipe () Int ()
primes n = upfrom 2 ‘connect‘ sieve ‘connect‘ take n
```

The *sieve* pipe is defined recursively, which adds a new *filter* and *connect* operation in each recursive call. So, as more primes are calculated, the number of *filter*/*connect* layers grows.

The `streamly` library has a different interface. Instead of creating stream transformers, we operate directly on a stream of data. We can adapt the previous algorithm as seen below. Note that reverse function composition (&) takes the role of the *connect* operator.

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

```
sieve :: (Monad m, IsStream t, MonadTrans t, Monad (t m)) ⇒ SerialT m Int → t m Int
sieve s = do
  mResult ← lift (uncons s)
  case mResult of
    Nothing → error "expected infinite stream"
    Just (p, s′) → p ‘cons‘ sieve (filter (λx → x ‘mod‘ p ≢ 0) s′)
primes :: (Monad m, IsStream t, MonadTrans t, Monad (t m)) ⇒ Int → t m Int
primes n = upfrom 2 & sieve & take n
```

We can see that both the `pipes` and `conduit` libraries, which use an ADT representation, show the quadratic performance behavior for a use case with a large number of connect steps. On the other hand, the continuation-based representation shows improved performance behavior. The `streamly` library is in the middle of the previous approaches for this benchmark. We suspect that the slight overhead of `proxyrep` compared to `contpipe` can be explained by the specialization to *IO* () in the latter type.

6.1.2 `deep-pipe` / `deep-seq` benchmarks

The next two benchmarks are micro-benchmarks of the `bind` and `connect` operations. The essence of these benchmarks is the following code:

```
iter :: Int → (a → a) → a → a
iter n f x = loop n x where
  loop k y = if k ≡ 0 then y else loop (k − 1) (f y)
skip∞ :: Pipe i o ()
skip∞ = forever skip
deepPipe :: Int → Pipe () Int x
deepPipe n = iter n (skip∞‘connect‘) (forever (yield 0))
deepSeq :: Int → Pipe () Int x
deepSeq n = iter n (≫skip∞) (forever (yield 0))
```

In these benchmarks, we construct *n* layers of `bind`/`connect` operations with a pipe forever outputting the number *0*. Spivey describes this intuitively in the following fashion:

```
deepSeq n =
  (...((forever (output 0)) ≫ skip∞) ≫ skip∞...) ≫ skip∞
deepPipe n =
  skip∞ ‘connect‘ (skip∞ ‘connect‘ (... ‘connect‘ (skip∞ ‘connect‘ forever (output 0))))
```

Finally, we take *n* elements from the stream of zeroes with ‘*connect*‘ *take n*.

The interface of `streamly` is incompatible with the structure of this benchmark. Because it is not as lazy as the other approaches creating a $skip_\infty$ stream results in an endless loop. Additionally, connecting with a $skip_\infty$ pipe has no meaningful counterpart in `streamly`. A close analogue is of course composing with the identity function, but this corresponds to connecting with a pipe which yields all inputs it receives. Hence, we have omitted `streamly` from these benchmarks.

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

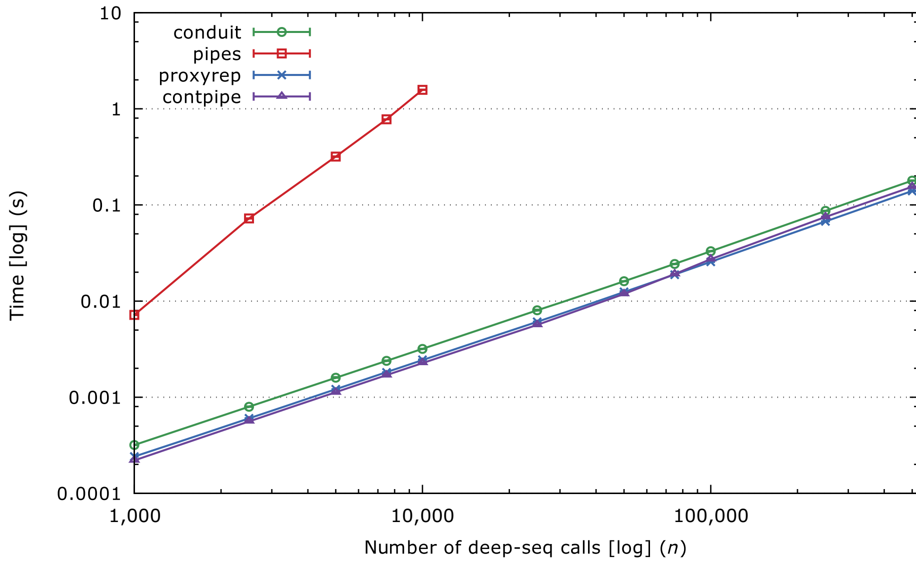

Fig. 2. Results of the `deep-seq` benchmark.

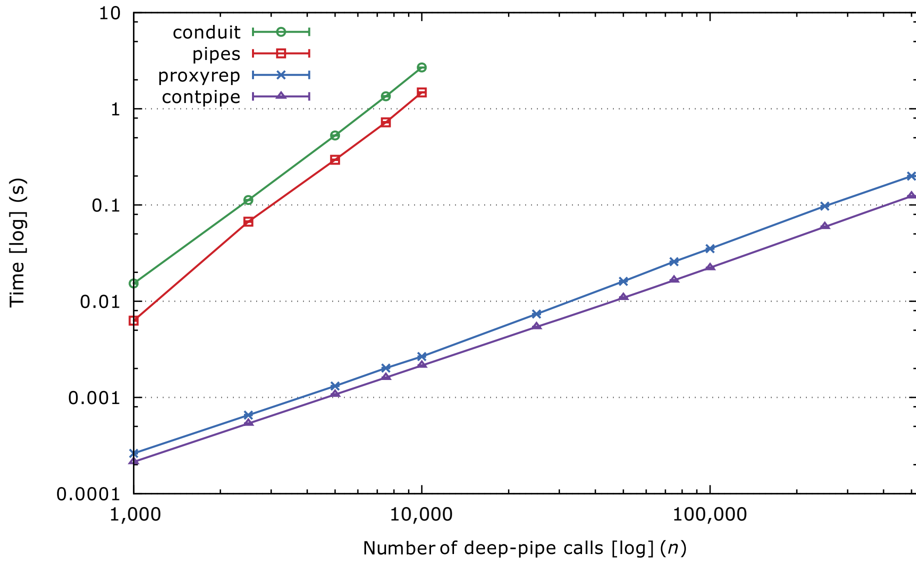

Fig. 3. Results of the `deep-pipe` benchmark.

The results of both benchmarks are shown in Figures 2 and 3. The x-axis shows the n parameter to the *deepSeq*/*deepPar* functions and the *take* n step. The y-axis shows the runtime (lower is better). Both axes are shown on a logarithmic scale to enhance readability.

In the `deep-seq` benchmark, we see that the `pipes` library shows the expected bad performance behavior for this use case. The `conduit` library uses a *Codensity* encoding

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

and thus does not suffer bad performance here. This is the behavior we expect from *Codensity* as described by Voigtländer (2008). The continuation-based approach also has similar performance to the *Codensity*-based approach.

In the `deep-pipe` benchmark, both the `pipes` and `conduit` libraries show bad performance behavior while the continuation-based representation does not. This is not surprising since it can be seen as a more extreme version of the `primes` benchmark.

6.2 Advertisement analytics benchmark

The previous benchmarks have been created to demonstrate the strengths of the continuation-based encoding, the *connect* operation. In practice, various other operations such as *map*, *filter*, or *fold* are used as well. In this benchmark, we look at a use case inspired by Chintapalli *et al.*. (2016), which was done by Yahoo to benchmark Java Virtual Machine-based distributed stream processing frameworks. This provides a possible real-world scenario for which these streaming libraries might be compared.

The idea behind the benchmark is to simulate an advertisement analytics pipeline. Various advertisement events (such as user clicks, views, and purchases) are stored in a message queue (Kafka). The `view` events need to be linked to advertisement campaign data stored in an in-memory store (Redis), and a counter is stored for each advertisement campaign/time window (also in Redis). The original use case measured the throughput of a continuously running stream processing application, also taking into account latecomers. We simplify this use case and measure the time needed to process up to 500,000 events in the Kafka queue.

A code sketch with explanatory comments for this use case is given below.

```
  -- setup Kafka/Redis connection
  -- countData represents a pipe producing the final data to write into redis,
  -- before counting the values for each key
let countData =          -- current type of produced values
  -- read messages from Kafka
  readKafka `connect`              -- KafkaRecord (Maybe ByteString, Maybe ByteString)
  -- read the value from the Kafka message
  map crValue `connect`            -- Maybe ByteString
  -- parse the value to JSON
  map parseJson `connect`          -- Maybe EventRecord
  -- remove Nothing values
  concat `connect`                 -- EventRecord
  -- only process 'view' events
  filter viewFilter `connect`      -- EventRecord
  -- fetch advertisement id and event time from record
  map eventProjection `connect`    -- (CampaignId, EventTime)
  -- use the advertisement id to fetch the campaign id from Redis
  mapM joinRedis `connect`         -- (CampaignId, AdId, EventTime)
  -- calculate the campaign time window
  map campaignTime `connect`       -- (CampaignId, WindowTime)
```

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

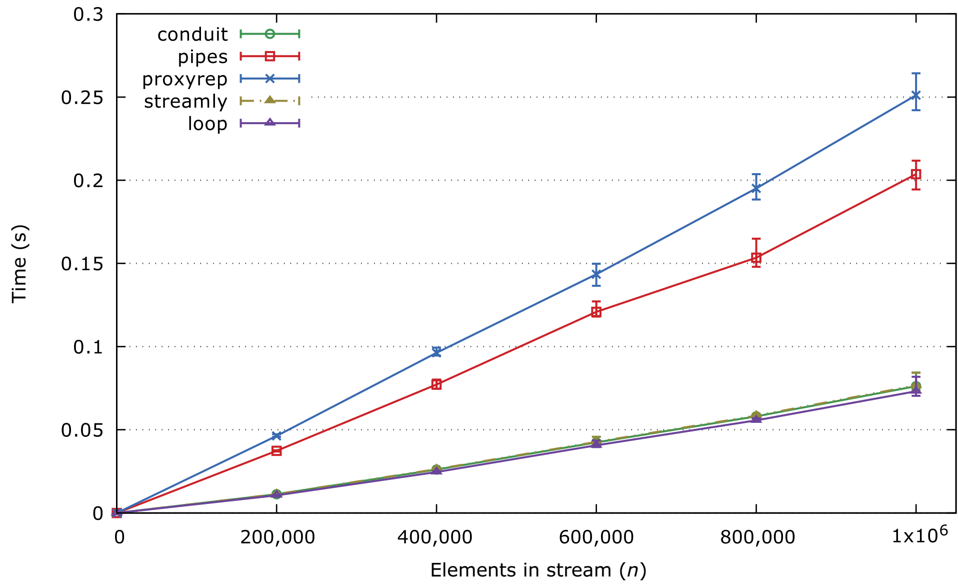

Fig. 4. Results of the map micro-benchmark.

-- only process *n* events
take n -- (CampaignId, WindowTime)
-- count the number of events by key: a tuple of the campaign id and time window
countByKey countData ‘*connect*‘
-- write data to Redis
mapM writeRedis -- ()
-- close Kafka/Redis connection

This code consists of a few *connect* steps interleaved with simple processing steps like *map* and *filter*. Near the end, it counts the values for each key, a *fold* over the stream, and writes the resulting data. The number of *connect* operations is relatively low, and other operations such as *map* and *filter* are important as well. For this reason, we first take a look at micro-benchmarks of these operations.

6.2.1 Micro-benchmarks

The advertisement analytics use case involves the *map*, *mapM*, *concat*, *filter*, and *fold* operations. We have taken benchmarks for these operations from the `streamly` benchmark suite (Kumar, 2018). All benchmarks follow the pattern of creating a simple stream and applying the operation under test on that stream. Our benchmarks differ from the `streamly` benchmark in that we force the evaluation of the full result list.

In the `map` benchmark, we do *map* (+ *1*) on a stream of numbers. In the `mapM` benchmark, we do *mapM return* on a stream of numbers. In the `filter` benchmark, we do *filter even* on a stream of numbers. In the `concat` benchmark, we do *concat* after doing *replicate 3* on a stream of numbers. In the `fold` benchmark, we calculate the sum of a stream of numbers.

The results for each of the operations are shown in, respectively, Figures 4, 5, 6, 7, and 8. All representations are benchmarked together with a loop version. This loop version

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

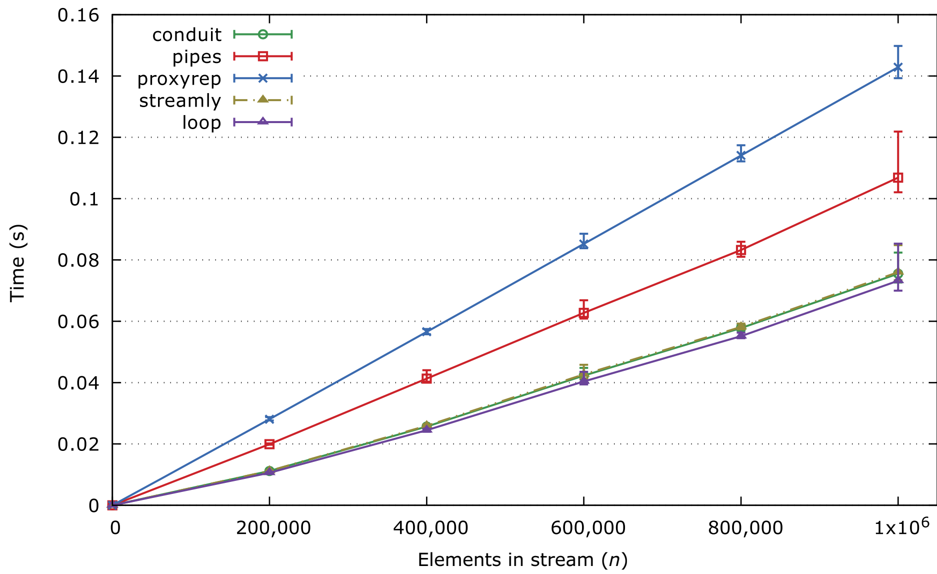

Fig. 5. Results of the mapM micro-benchmark.

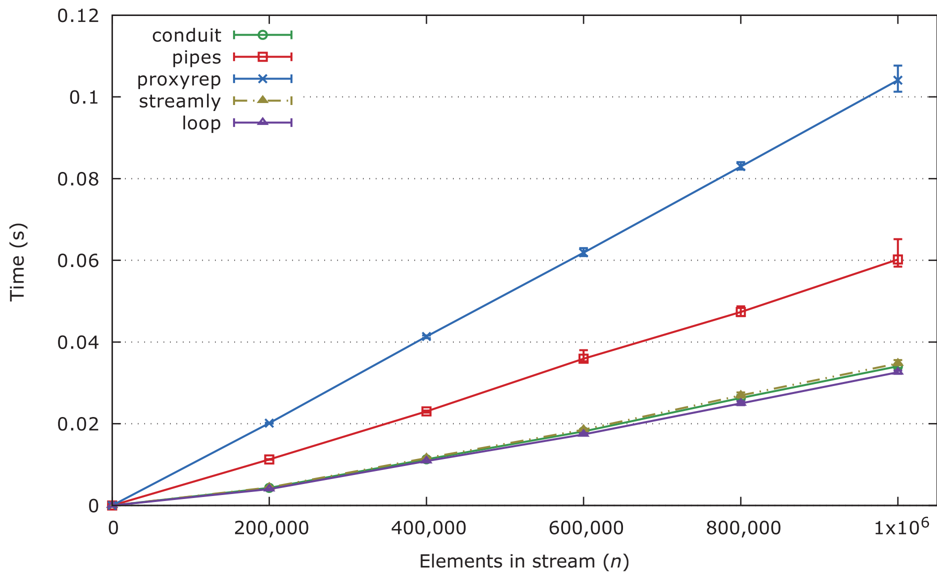

Fig. 6. Results of the filter micro-benchmark.

utilizes the same algorithm to create the stream of numbers in a tight loop. We can use this to see whether GHC was able to completely optimize the representation away with the help of the library's fusion rules. For example, the loop version of the map version is as follows:

$$mapBench :: Monad\ m \Rightarrow Int \rightarrow m\ [Int]$$
$$mapBench\ n = go\ [\,]\ n$$

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

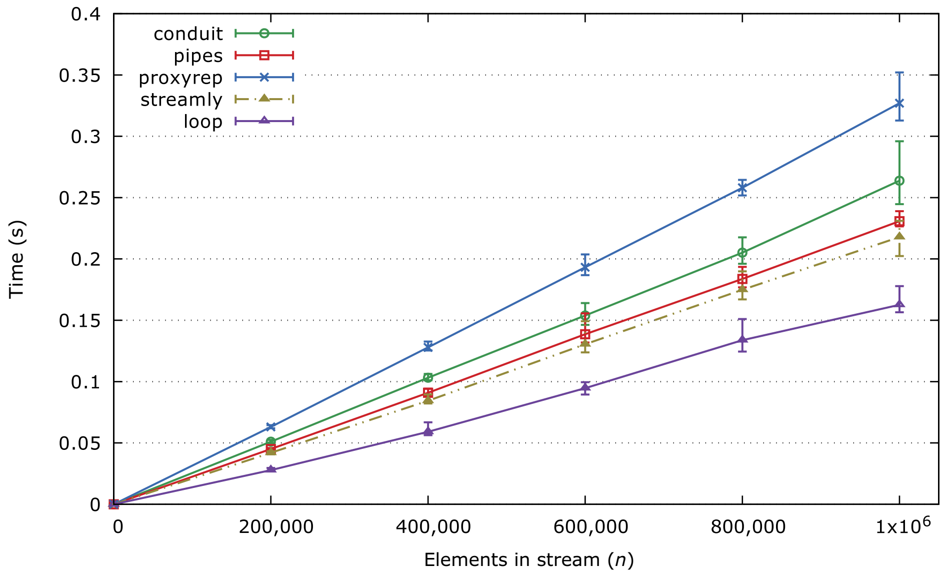

Fig. 7. Results of the concat micro-benchmark.

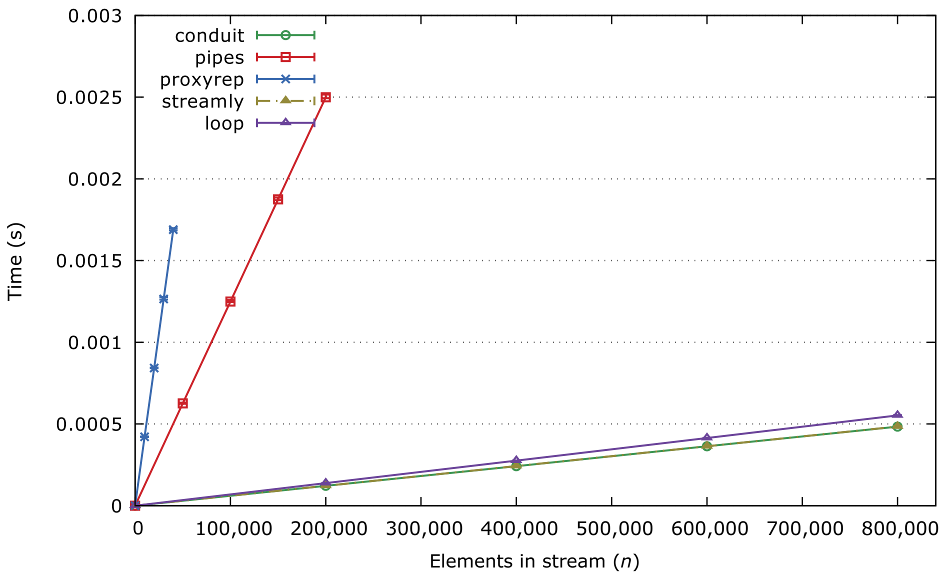

Fig. 8. Results of the fold micro-benchmark.

$$
\begin{aligned}
&\textbf{where} \\
&\quad go :: Monad\ m \Rightarrow [Int] \rightarrow Int \rightarrow m\ [Int] \\
&\quad go\ acc\ x \mid x < 0 = return\ acc \\
&\quad go\ acc\ x \qquad\quad = go\ ((x + 1) : acc)\ (x - 1)
\end{aligned}
$$

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

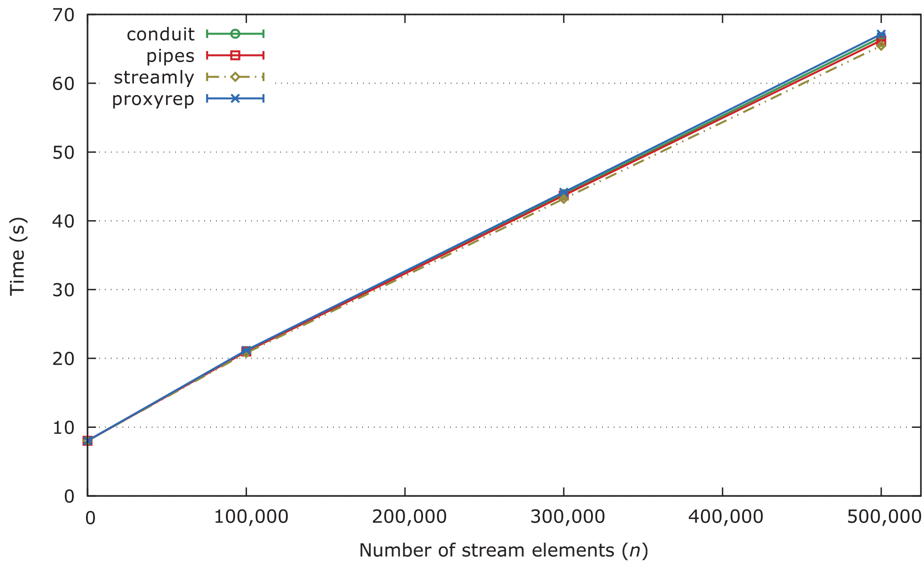

Fig. 9. Results of the `yahoo` benchmark.

Discussion. As with all micro-benchmarks, we have to consider what is actually being benchmarked here. First, a main aspect of the performance difference comes down to the library's fusion rules. As most of the benchmarks are small and fairly simple, we see that both `streamly` and `conduit` fuse to the loop version everywhere except for the *concat* benchmark. In those cases `pipes` performs a bit worse, since the creation and consumption of the *Proxy* representation still happens during runtime. The `proxyrep` representation performs worse than `pipes` due to the function representation which hinders certain GHC optimizations such as inlining within the representation, which leaves this extra work to be done at runtime. We utilize the same rewrite rules for `proxyrep` as `Pipes`, as we describe in Section 6.2.3, so it does not fuse to the loop version either.

In the `concat` benchmark, the `pipes` and `proxyrep` version compiles to a single loop, where the differences are again due to the function-based representation. The `conduit` version performs worse as it does not fuse into a single loop.

6.2.2 Advertisement analytics benchmark

Now, we move on to the actual benchmark of the use case scenario. Figure 9 shows the result of this benchmark. There is about 8 seconds of overhead for the Kafka connection, especially noticeable when processing only 1 element. This is due to various processes happening in Kafka, such as waiting for additional consumers to join in before sending data. We have not tweaked the Kafka configuration to reduce this delay and just consider it as a constant 8 second delay.

All of the libraries have similar performance behavior in this use case. The differences in performance present in the various micro-benchmarks are overshadowed by the main work done in the benchmark.

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

6.2.3 Remarks on optimization

The existing Haskell libraries (`pipes`, `conduit`, and `streamly`) have put effort into optimizing their performance. In this section, we discuss our efforts in porting some of their optimization techniques to the continuation-based encoding.

Inlining. Adding `INLINE` annotations makes GHC much more likely to inline the annotated function. Inlining functions typically has a positive impact on performance. We started by mimicking the annotations in the `pipes` library. Then, we followed the approach of the `streamly` benchmark suite (Kumar, 2018) and tried various compositions of adding and removing inline annotations and ultimately chose the composition which seemed to perform the best in the benchmark.

Rewrite Rules/Fusion. Rewrite rules (Peyton Jones *et al.*, 2001) enable the implementation of *fusion*, which fuses multiple costly steps traversing a structure into one traversal. In our case, it is interesting to combine multiple *connect* operations. For example, *p* ‘*connect*‘ *map* (+ *1*) ‘*connect*‘ *map* (+ *1*) can be fused into *p* ‘*connect*‘ *map* (+ *2*). To realize this, we have adapted the rewrite rules present in the `pipes` library (Gonzalez, 2014). Adapting these rules correctly required a few extra steps, which we discuss in the next two paragraphs. During development, the `inspection-testing` library by Breitner (2018) was a helpful tool to quickly test whether fusion was happening correctly.

The strategy of pipes’ rewrite rules is to transform as much of the pipeline as possible to a form which uses the *for* function. It takes an input pipe *p*, of type *Pipe i x a*, and a function *f*, of type $x \to$ *Pipe i o* (). Each *output x* in *p* is replaced by *f x*, thus resulting in a pipe with *o* outputs, or *Pipe i o a*. The implementation as seen in the `pipes` library is repeated below.

```
for :: Pipe i x a → (x → Pipe i o ()) → Pipe i o a
for p0 f = go p0 where
  go (Input h)    = Input (λx → go (h x))
  go (Output o r) = f o ≫ go r
  go (Return a)   = Return a
```

We elaborate on three rewrite rules, which enables the fusion of the earlier example. The first rule converts a *connect* followed by a *map* into a *for* which will replace each *yield x* with *yield* (*f x*). The second rule removes a *for* call when it is applied to a *yield x* pipe and replaces it with *f x*. The last rule reassociates the *for* operation.

```
"rule 1" ∀ · p f.p ‘connect‘ map f = for p (λy → yield (f y))
"rule 2" ∀ · x f.for (yield x) f = f x
"rule 3" ∀ · p f g.for (for p f) g = for p (λx → for (f x) g)
```

Using these three rules, GHC can fuse *p* ‘*connect*‘ *map* (+ *1*) ‘*connect*‘ *map* (+ *1*) into *p* ‘*connect*‘ *map* (+ *2*).

```
p ‘connect‘ map (+ 1) ‘connect‘ map (+ 1)
  = (rule 1 fires on left connect)
for p (λy → yield (y + 1)) ‘connect‘ map (+ 1)
```

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

```
    = (rule 1 fires on remaining connect)
  for (for p (λy → yield (y + 1))) (λy → yield (y + 1))
    = (rule 3 fires)
  for p (λx → for (yield (x + 1)) (λy → yield (y + 1)))
    = (rule 2 fires on inner for)
  for p (λx → yield (x + 2))
```

Which is indeed equivalent to *p* ‘*connect*‘ *map* (+ 2) by following rule 1 backwards. While porting these rules to the three-continuation encoding, we encountered two issues which we detail in the following paragraphs.

Implementation of *for*. First, when we try to implement the *for* function for the continuation-based encoding, we encounter a similar problem as with the *connect* function: we need access to a continuation which we do not know upfront. If we implement it assuming infinite pipes, we get the following implementation:

```
for :: ContPipe i m a → (m → ContPipe i o ()) → ContPipe i o a
for p0 f = MakePipe (λk k_i k_o → runPipe p0 err k_i (go k_i k_o))
  where
    go k_i k_o = MakeOutCont $ λo (MakeInCont g) →
      runPipe (f o) (λa k'_i k'_o → g (go k'_i k'_o)) k_i k_o
    err = error "terminated"
```

However, this implementation proved problematic in the advertisement analytics benchmark, so we were forced to rethink this implementation.

The challenge in implementing the function is as follows: in the location where *err* is passed, the interpretation of the *return* constructor, we need to access the k_o-continuation which was introduced *last* by *go*.

Based on this intuition and using impure *IO*, it is straightforward to implement this idea. At every step in *go*, we save the current k_o-continuation in an *IORef* and we read out the last continuation written in this way in the interpretation of the return continuation. The reference is initialized with the k_o-continuation, we are given from the *MakePipe* constructor.

```
for :: ContPipe i m a → (m → ContPipe i o ()) → ContPipe i o a
for p0 f = MakePipe $ λk k_i k_o → unsafePerformIO $ do
    ref_o ← newIORef k_o
    return $ runPipe p0
      (λa k'_i _ → unsafePerformIO $ do
        ok ← readIORef ref_o
        return (k a k'_i k_o)) k_i (go ref_o k_i k_o)
      where
        go ref_o k_i k_o = MakeOutCont $ λo (MakeInCont g) →
          runPipe (f o) (λa k'_i k'_o → unsafePerformIO $ do
            modifyIORef' ref_o (λ_ → k'_o)
            return (g (go ref_o k'_i k'_o))
          ) k_i k_o
```

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

Admittedly, this implementation is not idiomatic Haskell and conceptually we see no reason why a pure version of this idea is not possible, but we have not managed to implement it yet. Interesting to note is that an implementation of *connect* which works on returning pipes is possible based on this idea; its implementation can be found in Appendix B.

Eta-Expansion. Second, simply copying the rewrite rules from `pipes` does not make everything fuse fully by itself. In between applying the inline rules, GHC inserts a `newtype`-coercion of *ContPipe* and an eta-expansion, preventing the rules from firing after that point. If we look at the output of the `-ddump-rule-rewrites` option when looking at the p ‘*connect*‘ *map* (+ *1*) ‘*connect*‘ *map* (+ *1*) example, we see the following steps (GHC output is simplified for clarity):

$$
\begin{array}{l}
p \text{ `}\mathit{connect}\text{` } \mathit{map}\ (+\ 1)\ \text{`}\mathit{connect}\text{` } \mathit{map}\ (+\ 1) \\
\quad = (\mathit{rule}\ 1\ \mathit{fires\ on\ left\ connect}) \\
\mathit{for}\ p\ (\lambda y \rightarrow \mathit{yield}\ (y + 1))\ \text{`}\mathit{connect}\text{` } \mathit{map}\ (+\ 1) \\
\quad = (\mathbf{in}\ \mathit{between\ these\ rules}, \mathit{GHC\ eta} - \mathit{expands\ the}\ \mathbf{newtype}) \\
\mathit{for}\ p\ (\lambda y \rightarrow ((\lambda k\ k_i\ k_o \rightarrow (\mathit{yield}\ (y + 1)\ \text{`}\mathit{cast}\text{` } \ldots)\ k\ k_i\ k_o)\ \text{`}\mathit{cast}\text{` } \ldots)\ \text{`}\mathit{connect}\text{` } \mathit{map}\ (+\ 1) \\
\quad = (\mathit{rule}\ 1\ \mathit{fires\ on\ remaining\ connect}) \\
\mathit{for}\ (\mathit{for}\ p\ ((\lambda k\ k_i\ k_o \rightarrow (\mathit{yield}\ (y + 1)\ \text{`}\mathit{cast}\text{` } \ldots)\ k\ k_i\ k_o)\ \text{`}\mathit{cast}\text{` } \ldots))\ (\lambda y \rightarrow \mathit{yield}\ (y + 1)) \\
\quad = (\mathit{rule}\ 3\ \mathit{fires}) \\
\mathit{for}\ p\ (\lambda x \rightarrow \mathit{for}\ ((\lambda k\ k_i\ k_o \rightarrow (\mathit{yield}\ (y + 1)\ \text{`}\mathit{cast}\text{` } \ldots)\ k\ k_i\ k_o)\ \text{`}\mathit{cast}\text{` } \ldots)) \\
\qquad (\lambda y \rightarrow \mathit{yield}\ (y + 1))) \\
\quad (\mathit{rule}\ 2\ \mathit{does\ not\ fire})
\end{array}
$$

To fix this problem, we tested the GHC option `-fno-do-lambda-eta-expansion`, and replacing `newtype` with a strict `data` field. Both resolved the problem, but the latter achieved better performance in our benchmarks.

Fusion Framework. Both `conduit` and `streamly` have a more extensive fusion framework, which was visible in the micro-benchmarks. This framework allows the rewriting of a stream generation plus stream consumption into an optimized form using shortcut fusion. We leave the creation of such a framework for `proxyrep` as future work. Note that `pipes` would benefit from this effort as well since both libraries present the same API.

7 Related work

We have covered the main related works of Spivey (2017), Shivers and Shivers & Might (2006), and the `pipes` library (Gonzalez, 2012) in the body of the paper. Below we discuss additional related work.

Encodings. The Church (Church, 1941; Corrado & Alessandro, 1985) and Scott (Mogensen, 1992) encodings encode ADTs using functions. The encoding derived in this paper has a close connection to the Scott encoding. The Scott encoding for *Producer* and *Consumer* are *ScottP* and *ScottC*. By moving the quantified variable a to the definition, we obtain *SP* and *SC*.

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

$$
\begin{aligned}
&\textbf{newtype}\ ScottP\ o = ScottP\ (\forall a.(o \to ScottP\ o \to a) \to a) \\
&\textbf{newtype}\ ScottC\ i = ScottC\ (\forall a.((i \to ScottC\ i) \to a) \to a)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{newtype}\ SP\ o\ a = SP\ ((o \to SP\ o\ a \to a) \to a) \\
&\textbf{newtype}\ SC\ i\ a = SC\ (((i \to SC\ i\ a) \to a) \to a)
\end{aligned}
$$

Then, $\forall a.SP\ o\ a$ is representationally equivalent to $Producer_{Alt}$ and similarly for $\forall a.SC\ i\ a$ and $Consumer_{Alt}$ (see Appendix C).

If we look at the Scott encoding $ScottPipe_{\infty}$ for $Pipe_{\infty}$, we can obtain an equivalent representation to $Result_R$ by using SP and SC instead of $ScottPipe_{\infty}$ in the parameter corresponding to their operations.

$$
\begin{aligned}
&\textbf{newtype}\ ScottPipe_{\infty}\ i\ o = ScottPipe_{\infty} \\
&\quad (\forall a.(o \to ScottPipe_{\infty}\ i\ o \to a) \to ((i \to ScottPipe_{\infty}\ i\ o) \to a) \to a)
\end{aligned}
$$

$$
\textbf{type}\ SP_{\infty}\ i\ o = \forall a.(o \to SP\ o\ a \to a) \to ((i \to SC\ i\ a) \to a) \to a
$$

We dubbed this the *orthogonal encoding* due to the separation of the operations.

Conduit. The `conduit` library (Snoyman, 2011) is another popular choice for Haskell stream processing. The two main differing points of `conduit` with `pipes` are a built-in representation of leftovers and detection of upstream finalization. Leftovers are operations representing unprocessed outputs. For example in a *takeWhile* pipe, which takes outputs until a condition is matched, the first element not matching the condition will also be consumed. This element can then be emitted as a leftover, which is consumed by the downstream with priority. Detecting upstream finalization is handled by *input* returning *Maybe* values, where *Nothing* represents the finalization of the upstream.

Streamly. The `streamly` library (Kumar, 2017) takes a different approach to stream processing. As opposed to working with stream transformers, as seen in `conduit` or `pipes`, `streamly` works on the stream directly. In essence, this makes it a much more generalized version of working with (potentially infinite) lists. The library has been highly optimized for performance and is becoming more popular as an alternative library for stream processing.

Parsers. Spivey mentioned in his work (Spivey, 2017) that the *ContPipe* approach might be adapted to fit the use case of parallel parsers (Claessen, 2004). However, after gaining more insight into *ContPipe*, it does not seem that the connecting operation for parsers immediately fits the pattern presented in this paper. One of the problematic elements is the *fail* operation, which is not passed as-is to the newly connected structure, but given a non-trivial interpretation. Namely, an interpretation dependent on the other structure during the recursive connect process.

Shallow To Deep Handlers. The handlers framework by Kammar *et al.* (2013) supports both shallow handlers, based on case analysis, and deep handlers, based on folds. They cover an example of transforming a producer and consumer connecting function from

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

shallow handlers to deep handlers. This example is related to our simplified setting in Section 3. To do this, they introduce *Prod* and *Cons*, which are equivalent to our *ProdPar* and *ConsPar*. Compared to their example, we take a more step-by-step explanatory approach and additionally move to more complicated settings in our further sections.

Multihandlers. The Frank language (Lindley *et al.*, 2017) is based on shallow handlers and supports a feature called multihandlers. These handlers operate on multiple inputs which have uninterpreted operations, much like pattern matching on multiple free structures. The patterns we have handled in this paper are concerned with pattern matching on multiple data structures and a mutual relation between these functions. This seems like an interesting connection to investigate further.

Stream Optimizations. Hirzel *et al.* (2013) discuss stream processing optimizations used across a variety of disciplines. They list some commonly occurring optimization strategies: operator reordering, redundancy elimination, operator separation, fusion, etc. The fusion strategy is very common practice in Haskell by using GHC's rewrite rules (Peyton Jones *et al.*, 2001). We have similarly adopted fusion rules, by emulating the approach found in the `pipes` library (Gonzalez, 2012).

Stream Fusion. Coutts (2011) proposed stream fusion by building on previous work on both fold/build fusion (Gill *et al.*, 1993) and destroy/unfoldr fusion (Svenningsson, 2002). This is done by extending possibly terminating streams which either yield or terminate with the possibility of *skipping* to produce a value. This enables formulating, for example, *filter* in a non-recursive form, enabling fusion with such functions.

Stream Fusion with Staging. Kiselyov *et al.* (2017) transform stream processing pipelines to optimized code by using multi-stage programming, or staging for short. They implement their approach in OCaml and Scala using the respective staging facilities of these languages. The result of their work is a declarative stream processing library which transforms the specified pipelines into optimized imperative loop code.

Representation of Stream Functions. Ghani *et al.* (2009) represent continuous functions on streams in terms of a stream processing data type. This data type is roughly the following:

```
data SP i o = Get (i → SP i o)
            | Put o (SP i o)
```

This data type is *Pipe* from Section 1 where the *Return* constructor is left out. However, due to Haskell's mixing of inductive and co-inductive data types, a subtlety is lost. The recursive *SP i o* in the *Get* constructor is inductive: only a finite amount of input may be read until a *Put* occurs, while the recursive *SP i o* in the *Put* constructor is co-inductive: there can be an infinite amount of *Puts*. This ensures that the stream processor is productive, from every point a finite amount of steps are needed to reach the next *Put* constructor.

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

8 Conclusion

This paper gives an in-depth explanation of the principles behind the fast connecting of the continuation-based encoding introduced by Shivers & Might (2006). We first derive the new representation from the ADT implementation in a simplified setting of one-sided pipes, then we represent pipes using this alternative representation. We adapt this new representation in the original setting of two-sided pipes and comment on the issues of adding the return operation.

We derive this alternative encoding for bidirectional pipes, as in the `pipes` library. This results in a more general version of this representation and retains the performance of its efficient connect implementation. This derivation is also a simple example of how this pattern of derivation can be applied in a different situation.

The results of applying Spivey's benchmarks (Spivey, 2017) are similar to Spivey's original results, meaning that the generalized encoding has retained the performance characteristics. We also include various popular stream processing libraries, namely `pipes`, `conduit`, and `streamly`. In addition to Spivey's benchmarks, we also take a look at Yahoo's benchmark (Chintapalli *et al.*, 2016) of an advertisement analytics use case and micro-benchmarks of the operations involved therein.

We describe how we ported several techniques for improving performance, inspired by the conventional stream processing libraries. In addition, we discuss several difficulties we encountered during the implementation of these improvements. The continuation-based encoding is as performant as the other common streaming libraries in the advertisement analytics use case. The encoding performs worse than the other libraries in the micro-benchmarks, which could be alleviated by a more advanced fusion framework.

The continuation-based encoding described in this paper has been made available as a library on github (Pieters, 2018b).

Acknowledgements

We would like to thank Nicolas Wu, Alexander Vandenbroucke, and the anonymous reviewers for their feedback. This work was partly funded by the Flemish Fund for Scientific Research (FWO).

Supplementary materials

To view supplementary material for this article, please visit https://doi.org/10.1017/S0956796820000192

Conflicts of Interest

None.

References

Blöndal, B., Löh, A. & Scott, R. (2018) Deriving via: or, how to turn hand-written instances into an anti-pattern. In Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27–17, 2018, pp. 55–67.

Breitner, J. (2018). A promise checked is a promise kept: Inspection testing. In Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell. Haskell 2018. New York, NY, USA: ACM, pp. 14–25.

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

Chintapalli, S., Dagit, D., Evans, B., Farivar, R., Graves, T., Holderbaugh, M., Liu, Z., Nusbaum, K., Patil, K., Peng, B. J. & Poulosky, P. (2016) Benchmarking streaming computation engines: Storm, flink and spark streaming. In 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1789–1792.

Church, A. (1941) *The Calculi of Lambda-Conversion*. Princeton, New York: Princeton University Press.

Claessen, K. (2004) Parallel parsing processes. *J. Funct. Program.* **14**(6), 741–757.

Corrado, B. & Alessandro, B. (1985) Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comput. Sci.* **39**, 135–154.

Coutts, D. (2011) *Stream Fusion: Practical Shortcut Fusion for Coinductive Sequence Types*. PhD Thesis, UK: University of Oxford.

Ghani, N., Hancock, P. & Pattinson, D. (2009) Representations of stream processors using nested fixed points. *Log. Methods Comput. Sci.* **5**(3).

Ghani, N., Uustalu, T. & Vene, V. (2004) Build, augment and destroy, universally. In *Programming Languages and Systems*, Chin, W.-N. (ed). Berlin, Heidelberg: Springer, pp. 327–347.

Gill, A. J., Launchbury, J. & Peyton Jones, S. L. (1993) A short cut to deforestation. In Proceedings of the Conference on Functional programming Languages and Computer Architecture, FPCA 1993, Copenhagen, Denmark, June 9–11, 1993, pp. 223–232.

Gonzalez, G. (2012) *Haskell Pipes Library*. http://hackage.haskell.org/package/pipes.

Gonzalez, G. (2014) *Stream Fusion for Pipes*. http://haskellforall.com/2014/01/stream-fusion-for-pipes.html.

Hirzel, M., Soulé, R., Schneider, S., Gedik, B. & Grimm, R. (2013) A catalog of stream processing optimizations. *ACM Comput. Surv.* **46**(4), 46:1–46:34.

Kammar, O., Lindley, S. & Oury, N. (2013) Handlers in action. In Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. ICFP'13. New York, NY, USA: ACM, pp. 145–158.

Kiselyov, O., Biboudis, A., Palladinos, N. & Smaragdakis, Y. (2017) Stream fusion, to completeness. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18–20, 2017, pp. 285–299.

Kumar, H. (2017) *Haskell Streamly Library*. http://hackage.haskell.org/package/streamly.

Kumar, H. (2018) *Streamly Benchmarks*. https://github.com/composewell/streaming-benchmarks.

Lindley, S., McBride, C. & McLaughlin, C. (2017) Do be do be do. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. POPL 2017. New York, NY, USA: ACM, pp. 500–514.

Mogensen, T. (1992) Efficient self-interpretation in lambda calculus. *J. Funct. Program.* **2**(3), 345–364.

O'Sullivan, B. (2009) *Haskell Criterion Library*. http://hackage.haskell.org/package/criterion.

Peyton Jones, S. L., Tolmach, A. & Hoare, T. (2001) Playing by the rules: Rewriting as a practical optimisation technique in ghc. In 2001 Haskell Workshop.

Pieters, R. P. (2018a) *Faster Coroutine Pipelines: A Reconstruction, Benchmarking Code*. https://github.com/rubenpieters/orth-pipes-bench.

Pieters, R. P. (2018b) *Faster Coroutine Pipelines: A Reconstruction, Library*. https://github.com/rubenpieters/Orthogonal-Pipes.

Pieters, R. P. & Schrijvers, T. (2019) Faster coroutine pipelines: A reconstruction. In *Practical Aspects of Declarative Languages*, Alferes, J. J. & Johansson, M. (eds), Cham: Springer International Publishing, pp. 133–149.

Plotkin, G. D. & Abadi, M. (1993) A logic for parametric polymorphism. In Proceedings of the International Conference on Typed Lambda Calculi and Applications. TLCA'93. London, UK, UK: Springer-Verlag, pp. 361–375.

Shivers, O. & Might, M. (2006) Continuations and transducer composition. In Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI'06. New York, NY, USA: ACM, pp. 295–307.

Snoyman, M. (2011) *Haskell Conduit Library*. http://hackage.haskell.org/package/conduit.

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

Spivey, M. (2017) Faster coroutine pipelines. *Proc. ACM Program. Lang.* **1**(ICFP), 5:1–5:23.
Svenningsson, J. (2002) Shortcut fusion for accumulating parameters & zip-like functions. In Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP'02), Pittsburgh, Pennsylvania, USA, October 4–6, 2002, pp. 124–132.
Voigtländer, J. (2008) Asymptotic improvement of computations over free monads. In Proceedings of the 9th International Conference on Mathematics of Program Construction. MPC'08. Berlin, Heidelberg: Springer-Verlag, pp. 388–403.

A Appendix

Both producer representations are isomorphic

We show the *Producer* case, an analogous strategy works for *Consumer*.

Theorem A.1. *The types $Producer_{Alt}\ o$ and $Producer\ o$ are isomorphic.*

Proof We show this by using $\mu x.\ G[x] = \forall x.(G[x] \to x) \to x$ (Plotkin & Abadi, 1993).

$$
\begin{array}{ll}
 & Producer_{Alt}\ o \\
= & \quad \text{(see Appendix C)} \\
 & \forall r.\ SP\ o\ r \\
= & \quad \text{(def. } SP\text{)} \\
 & \forall r. \mu x.\ (o \to x \to r) \to r \\
= & \quad \text{(property } \mu\text{)} \\
 & \forall r. \forall x.\ (((o \to x \to r) \to r) \to x) \to x \\
= & \quad \text{(swap } \forall\text{)} \\
 & \forall x. \forall r.\ (((o \to x \to r) \to r) \to x) \to x \\
= & \quad \text{(property } \mu\text{)} \\
 & \mu x. \forall r.\ (o \to x \to r) \to r \\
= & \quad \text{(uncurry)} \\
 & \mu x. \forall r.\ ((o, x) \to r) \to r \\
= & \quad \text{(property } \mu\text{)} \\
 & \mu x. \mu r.\ (o, x) \\
= & \quad \text{(drop unused } \mu\text{)} \\
 & \mu x.\ (o, x) \\
= & \quad \text{(def. } Producer\text{)} \\
 & Producer\ o
\end{array}
$$

□

An isomorphism consists of two inverses, here mr and mr^{-1}, for which $mr^{-1} \circ mr = id$ and $mr \circ mr^{-1} = id$ holds. Where mr is defined as:

$$
\begin{array}{l}
mr :: Producer\ x \to Producer_{Alt}\ x \\
mr\ (Producer\ o\ r)\ (ConsPar\ h) = h\ o\ (ProdPar\ (mr\ r))
\end{array}
$$

And mr^{-1} is defined as:

$$
\begin{array}{l}
mr^{-1} :: Producer_{Alt}\ x \to Producer\ x \\
mr^{-1}\ f = f\ (ConsPar\ (\lambda x\ p \to Producer\ x\ (helpermr^{-1}\ p)))
\end{array}
$$

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

$$helpermr^{-1} :: ProdPar\ x\ (Producer\ x) \to Producer\ x$$
$$helpermr^{-1}\ (ProdPar\ f) = f\ (ConsPar\ (\lambda x\ prod \to Producer\ x\ (helpermr^{-1}\ prod)))$$

We prove the first equation $mr^{-1} \circ mr = id$.

Theorem A.2. $mr^{-1} \circ mr = id$.

Proof We show this by equational reasoning.

$$
\begin{array}{ll}
& mr^{-1} \circ mr \\
= & \quad \text{(eta-expansion)} \\
& \lambda x \to mr^{-1}\ (mr\ x) \\
= & \quad \text{(def. } mr\text{)} \\
& \lambda x \to mr^{-1}\ (\mathbf{case}\ x\ \mathbf{of}\ (Producer\ o\ r) \to \lambda(ConsPar\ h) \to h\ o\ (ProdPar\ (mr\ r))) \\
= & \quad \text{(case analysis on } x\text{)} \\
& \lambda(Producer\ o\ r) \to mr^{-1}\ (\lambda(ConsPar\ h) \to h\ o\ (ProdPar\ (mr\ r))) \\
= & \quad \text{(def. } mr^{-1}\text{)} \\
& \lambda(Producer\ o\ r) \to (\lambda(ConsPar\ h) \to \\
& \quad h\ o\ (ProdPar\ (mr\ r)))\ (ConsPar\ (\lambda x\ p \to Producer\ x\ (helpermr^{-1}\ p))) \\
= & \quad \text{(beta-reduction)} \\
& \lambda(Producer\ o\ r) \to Producer\ o\ (helpermr^{-1}\ (ProdPar\ (mr\ r))) \\
= & \quad \text{(def. } helpermr^{-1}\text{)} \\
& \lambda(Producer\ o\ r) \to Producer\ o\ (mr\ r\ (ConsPar\ (\lambda x\ prod \to Producer\ x\ (helpermr^{-1}\ prod)))) \\
= & \quad \text{(def. } mr^{-1}\text{)} \\
& \lambda(Producer\ o\ r) \to Producer\ o\ (mr^{-1}\ (mr\ r)) \\
= & \quad \text{(coinduction hypothesis)} \\
& \lambda(Producer\ o\ r) \to Producer\ o\ r \\
= & \quad \text{(def. } id\text{)} \\
& id
\end{array}
$$

□

The second equation $mr \circ mr^{-1} = id$ is more complicated since we cannot do case analysis on $Producer_{Alt}$, which is the function type $\forall a.ConsPar\ o\ a \to a$. We assume that values of this type can always be written using the constructor function *output*. Where *output* is defined as:

$$output :: o \to Producer_{Alt}\ o \to Producer_{Alt}\ o$$
$$output\ o\ prod = \lambda(ConsPar\ cons) \to cons\ o\ (ProdPar\ prod)$$

Conjecture A.1. *Values of $Producer_{Alt}$ can always be written in the form output o r.*

This allows us to do a similar step to case analysis and complete the proof.

Theorem A.3. $mr \circ mr^{-1} = id$.

Proof We show this by equational reasoning.

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

$$
\begin{array}{ll}
& mr \circ mr^{-1} \\
= & \quad \text{(eta-expansion)} \\
& \lambda x \to mr\,(mr^{-1}\,x) \\
= & \quad \text{(def. } mr^{-1}) \\
& \lambda x \to mr\,(x\,(ConsPar\,(\lambda x\,p \to Producer\,x\,(helpermr^{-1}\,p)))) \\
= & \quad \text{(Conjecture A.1)} \\
& \lambda(output\,o\,r) \to mr\,((output\,o\,r)\,(ConsPar\,(\lambda x\,p \to Producer\,x\,(helpermr^{-1}\,p)))) \\
= & \quad \text{(def. } output) \\
& \lambda(output\,o\,r) \to mr\,((\lambda(ConsPar\,cons) \to \\
& \quad cons\,o\,(ProdPar\,r))\,(ConsPar\,(\lambda x\,p \to Producer\,x\,(helpermr^{-1}\,p)))) \\
= & \quad \text{(beta-reduction)} \\
& \lambda(output\,o\,r) \to mr\,((\lambda x\,p \to Producer\,x\,(helpermr^{-1}\,p))\,o\,(ProdPar\,r)) \\
= & \quad \text{(beta-reduction)} \\
& \lambda(output\,o\,r) \to mr\,(Producer\,o\,(helpermr^{-1}\,(ProdPar\,r))) \\
= & \quad \text{(def. } helpermr^{-1}) \\
& \lambda(output\,o\,r) \to mr\,(Producer\,o\,(r\,(ConsPar\,(\lambda x\,prod \to Producer\,x\,(helpermr^{-1}\,prod))))) \\
= & \quad \text{(def. } mr^{-1}) \\
& \lambda(output\,o\,r) \to mr\,(Producer\,o\,(mr^{-1}\,r)) \\
= & \quad \text{(def. } mr) \\
& \lambda(output\,o\,r) \to \lambda(ConsPar\,h) \to h\,o\,(ProdPar\,(mr\,(mr^{-1}\,r))) \\
= & \quad \text{(coinduction hypothesis)} \\
& \lambda(output\,o\,r) \to \lambda(ConsPar\,h) \to h\,o\,(ProdPar\,r) \\
= & \quad \text{(def. } output) \\
& \lambda(output\,o\,r) \to output\,o\,r \\
= & \quad \text{(def. } id) \\
& id
\end{array}
$$

□

B Appendix

Connect implementation

Connect implementation which also handles the return case by keeping track of the left *InCont* continuation and the right *OutCont* continuation impurely in *IORef* s. Then, the appropriate continuation is read in the interpretation of the return continuation.

```
connect' :: ContPipe i x m a → ContPipe x o m a → ContPipe i o m a
connect' p q = MakePipe $ λk k_i k_o → unsafePerformIO $ do
  ref_o ← newIORef k_o
  ref_i ← newIORef k_i
  let p' = runPipe p (λa ik' _ → unsafePerformIO $ do ok' ← readIORef ref_o; return (k a ik' ok'))
      q' = runPipe q (λa _ ok' → unsafePerformIO $ do ik' ← readIORef ref_i; return (k a ik' ok'))
      modp ik ok = p' (keepRefI ref_i ik) ok
      modq ik ok = q' ik (keepRefO ref_o ok)
  return $ modq (MakeInCont (λk'_o → modp k_i k'_o)) k_o
  where
    keepRefI :: IORef (InCont i a) → InCont i a → InCont i a
    keepRefI ref_i (MakeInCont f) = MakeInCont $ λ(MakeOutCont g) → f $ MakeOutCont $ λi ik →
      unsafePerformIO $ do
```

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press

```
      modifyIORef' ref_i (λ_ → ik)
      return (g i (keepRefI ref_i ik))
keepRefO :: IORef (OutCont o a) → OutCont o a → OutCont o a
keepRefO ref_o (MakeOutCont f) = MakeOutCont $ λo (MakeInCont g) → f o $ MakeInCont $ λok →
   unsafePerformIO $ do
      modifyIORef' ref_o (λ_ → ok)
      return (g (keepRefO ref_o ok))
```

C Appendix

Relation to scott encoding

We show the *Producer* case, an analogous strategy works for *Consumer*.

Theorem C.1. *The representation $Producer_{Alt}$ is $\sim_R$ (representationally equivalent) to $\forall a.SP\ o\ a$.*

The relevant definitions for the proof are

```
newtype ConsPar x a = ConsPar (x → ProdPar x a → a)
newtype ProdPar x a = ProdPar (ConsPar x a → a)
newtype SP o a = SP ((o → SP o a → a) → a)
type Producer_Alt o = ∀a.ConsPar o a → a
```

Proof First, we show that $ConsPar\ o\ a \to a \sim_R SP\ o\ a$.

$$
\begin{aligned}
& ConsPar\ o\ a \to a \\
\sim_R\ & \text{(def. } ConsPar) \\
& (o \to ProdPar\ o\ a \to a) \to a \\
\sim_R\ & \text{(def. } ProdPar) \\
& (o \to (ConsPar\ o\ a \to a) \to a) \to a \\
\sim_R\ & \text{(coinduction hypothesis)} \\
& (o \to SP\ o\ a \to a) \to a \\
\sim_R\ & \text{(def. } SP) \\
& SP\ o\ a
\end{aligned}
$$

Then,

$$
\begin{aligned}
& Producer_{Alt}\ o \\
\sim_R\ & \text{(def. } Producer_{Alt}) \\
& \forall a.ConsPar\ o\ a \to a \\
\sim_R\ & \text{(proven above)} \\
& \forall a.SP\ o\ a
\end{aligned}
$$

□

https://doi.org/10.1017/S0956796820000192 Published online by Cambridge University Press