Faster Coroutine Pipelines: A Reconstruction

Ruben P. Pieters ✉[0000−0003−0537−9403] and Tom Schrijvers[0000−0001−8771−5559]

KU Leuven, 3001 Leuven, Belgium
{ruben.pieters, tom.schrijvers}@cs.kuleuven.be

Abstract. Spivey has recently presented a novel functional representation that supports the efficient composition, or *merging*, of coroutine pipelines for processing streams of data. This representation was inspired by Shivers and Might's three-continuation approach and is shown to be equivalent to a simple yet inefficient executable specification. Unfortunately, neither Shivers and Might's original work nor the equivalence proof sheds much light on the underlying principles allowing the derivation of this efficient representation from its specification.
This paper gives the missing insight by reconstructing a systematic derivation in terms of known transformation steps from the simple specification to the efficient representation. This derivation sheds light on the limitations of the representation and on its applicability to other settings. In particular, it has enabled us to obtain a similar representation for pipes featuring two-way communication, similar to the Haskell `pipes` library. Our benchmarks confirm that this two-way representation retains the same improved performance characteristics.

Keywords: Stream Processing · Structured Recursion · Algebra.

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` [**?**] and `conduit` [**?**]. Common to both libraries is their representation of pipelines by an algebraic data type (ADT).

Spivey [**?**] 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 [**?**] and exhibits a very efficient *merge* 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 represenation for the two-way pipes featured in the `pipes` library.

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 the merging of "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 merge operator for the bidirectional three-continuation approach improves upon the `pipes` library's performance.

The rest of this paper is organized as follows. Section **??** briefly introduces both the ADT pipes encoding and the three-continuation approach. Section **??** derives the fast merging operation for a simplified setting. Section **??** derives the fast merging operation for the original pipe setting. Section **??** extends Spivey's approach with the bidirectional pipes operations. Section **??** presents the results of the primes benchmark by Spivey, on the approaches discussed in this paper. Section **??** discusses related work and Section **??** concludes this paper. The appendix is included in the extended version[1].

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. On the other hand, a bidirectional pipe additionally carries an output value when receiving values and an input value when outputting values. 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 datatype exhibits a monadic structure where the bind operation ($\gg=$) grafts one syntax tree onto another.

[1] `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW715.abs.html`

```
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: $input_P$: a pipe returning its received input, $output_P$: a pipe outputting a set value and $return_P$: a pipe returning a set value.

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

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

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

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

```
doubler_P :: Pipe Int Int a
doubler_P = do i ← input_P; output_P (i * 2); doubler_P
```

Another essential way of combining pipes is *merging* them. This connects the outputs of the upstream to the inputs of the downstream. In the implementation, $merge_{PL}$ performs a case analysis on the downstream q: if it is trying to output then that is kept and we keep searching, if it finds an input then we call $merge_{PR}$ on the wrapped continuation and the upstream. Then, in $merge_{PR}$ 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 and the merging process starts again. If at any point we see a return, then the merge finishes with this resulting return value. The implementation is given below.

```
merge_P :: Pipe i m a → Pipe m o a → Pipe i o a
merge_P p q = merge_PL q p where
  merge_PL :: Pipe m o a → Pipe i m a → Pipe i o a
  merge_PL (Input h)      p = merge_PR p h
  merge_PL (Output o r) p = Output o (merge_PL r p)
  merge_PL (Return a)    p = Return a

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

The merge operator enables expressing the merge of $doubler_P$ with itself. In this example the left $doubler_P$ is the upstream and the right $doubler_P$ is the

downstream. The result of this merge is a pipe which outputs the quadruple of its incoming values.

```
quadrupler_P :: Pipe Int Int a
quadrupler_P = doubler_P `merge_P` doubler_P
```

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

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

An example where we input *10*, receive *40* and then exit, is shown below.

```
λ > toIO_P quadrupler_P
10 ⟨Return⟩
out : 40
⟨Ctrl+C⟩
```

2.2 Three-Continuation Approach

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

Spivey has introduced *ContPipe*, a different pipe representation which enables a faster merge implementation [**?**]. It featuresthree continuations, one for each constructor. The first continuation $(a \to Result\ i\ o)$ represents the return constructor. The next two continuations, *InCont* i and *OutCont* o as part of *Result* i o, represent the input and output constructor respectively.

```
newtype ContPipe i o a =
  MakePipe {runPipe :: (a → Result i o) → Result i o}
type Result i o = InCont i → OutCont o → IO ()
newtype InCont i =
  MakeInCont {resume_I :: OutCont i → IO ()}
newtype OutCont o =
  MakeOutCont {resume_O :: o → InCont o → IO ()}
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_CP :: a → ContPipe i o a
return_CP a = MakePipe (λk k_i k_o → k a k_i k_o)

input_CP :: ContPipe i o i
input_CP = MakePipe (λk k_i k_o →
  resume_I k_i (MakeOutCont (λi k_i' → k i k_i' k_o)))

output_CP :: o → ContPipe i o ()
output_CP o = MakePipe (λk k_i k_o →
  resume_O k_o o (MakeInCont (λk_o' → k () k_i k_o')))
```

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

```
doubler_CP :: ContPipe Int Int a
doubler_CP = do i ← input_CP; output_CP (i * 2); doubler_CP
```

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

```
toIO_CP :: (Read i, Show o) ⇒ ContPipe i o () → IO ()
toIO_CP p = runPipe p (λ() _ _ → return ()) k_i k_o where
  k_i = MakeInCont (λk_o → do x ← readLn; resume_O k_o x k_i)
  k_o = MakeOutCont (λo k_i →
    do putStrLn ("out: " ++ show o); resume_I k_i k_o)
```

The merge function for *ContPipe* is defined as:

```
merge_CP p q = MakePipe (λk k_i k_o →
  runPipe q err (MakeInCont (λk_o' → runPipe p err k_i k_o')) k_o)
  where err = error "terminated"
```

With the merge definition we are able to create the quadrupler pipe as before. Running $toIO_{CP}\ quadrupler_{CP}$ results an identical scenario to the *Pipe* scenario from the previous section.

```
quadrupler_CP :: ContPipe Int Int a
quadrupler_CP = doubler_CP `merge_CP` doubler_CP
```

While Spivey has demonstrated the remarkable performance advantage of this merge 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.

3 Fast Merging for One-Sided Pipes

To offer a firmer grip on the problem, this section considers a simplified setting where pipes are one-sided, either only producing or only consuming data. For example, the *doubler* component can not be defined in this setting. The simplified setting gives a more straightforward path to the fast merging approach, which we generalize back to regular 'mixed' pipes in Section **??**.

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

If we specialize $merge_P$ for a *Consumer* and a *Producer*, we get:

```
merge_A :: Producer b → Consumer b → a
merge_A p q = merge_AL q p where
  merge_AL :: Consumer b → Producer b → a
  merge_AL (Consumer h) p = merge_AR p h
  merge_AR :: Producer b → (b → Consumer b) → a
  merge_AR (Producer o r) h = merge_AL (h o) r
```

3.2 Mutual Recursion Elimination

The two auxiliary functions $merge_{AL}$ and $merge_{AR}$ 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 $merge_{AL}$ and $merge_{AR}$ with respect to their additional parameter in a way that removes the term-level mutual recusion between them. Consider $merge_{AL}$ which does not use its parameter p directly, but only its interpretation by function $merge_{AR}$. We refactor this code to a form where $merge_{AR}$ has already been applied to p before it is passed to $merge_{AL}$. This adapted $merge_{AL}$ would then have type *Consumer* $b \to$ (*ConsPar'* $b \to a$) $\to a$. At the same time we apply a similar transformation to $merge_{AR}$, moving the application of $merge_{AL}$ 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 merge is then defined by appropriately placing newtype (un-)wrappers.

```
merge_Par :: Producer b → Consumer b → a
merge_Par p q = ml q (ProdPar (mr p)) where
  ml :: Consumer b → ProdPar b a → a
  ml (Consumer h) (ProdPar p) = p (ConsPar (λi → (ml (h i))))
  mr :: Producer b → ConsPar b a → a
  mr (Producer 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
alg_L f = λ(ProdPar prod) → prod (ConsPar f)
type Carrier_R o a = ConsPar o a → a
alg_R :: o → Carrier_R o a → Carrier_R o a
alg_R o prod = λ(ConsPar cons) → 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 $f\ r \to r$. This action denotes the computation performed at each node of the recursive datatype, 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 $(i \to r) \to r$ and $o \to r \to r$. Their definitions are:

```
fold_P :: (o → r → r) → Producer o → r
fold_P alg (Producer o r) = alg o (fold_P alg r)
fold_C :: ((i → r) → r) → Consumer i → r
fold_C alg (Consumer h) = alg (λi → fold_C alg (h i))
```

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 $merge_{Par}$ with folds using alg_L and alg_R.

```
merge_fold :: Producer x → Consumer x → a
merge_fold p q = fold_C alg_L q (ProdPar (fold_P alg_R p))
```

3.4 A Short-Cut to a Merge-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 constructor occurrences—this is known as *build* form. The *build* function then instantiates the abstracted constructors with the actual constructors; for *Consumer* and *Producer* they are:

$$
\begin{aligned}
&build_C :: (\forall r.((i \to r) \to r) \to r) \to Consumer\ i \\
&build_C\ g = g\ Consumer \\
&build_P :: (\forall r.(o \to r \to r) \to r) \to Producer\ o \\
&build_P\ g = g\ Producer
\end{aligned}
$$

For instance,

$$
\begin{aligned}
&prodFrom :: Integer \to Producer\ Integer \\
&prodFrom\ n = Producer\ n\ (prodFrom\ (n + 1))
\end{aligned}
$$

can be written as:

$$
\begin{aligned}
&prodFrom\ n = build_P\ (prodFrom'\ n)\ \textbf{where} \\
&\quad prodFrom' :: Integer \to (\forall r.(Integer \to r \to r) \to r) \\
&\quad prodFrom'\ n\ p = go\ n\ \textbf{where}\ go\ n = p\ n\ (go\ (n + 1))
\end{aligned}
$$

The motivation for these build functions is use of the fold/build fusion rule, a special form of short-cut fusion [**?**]. This rule can be applied when a fold directly follows a build, specifically for *Consumer* and *Producer* these fusion rules are:

$$
\begin{aligned}
fold_C\ alg\ (build_C\ cons) &= cons\ alg \\
fold_P\ alg\ (build_P\ prod) &= prod\ alg
\end{aligned}
$$

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 $merge_{fold}$. We can directly represent consumers and producers in terms of the carrier types of those two folds,

$$
\begin{aligned}
&\textbf{type}\ Consumer_{Alt}\ i\ = \forall a.Carrier_L\ i\ a \quad \text{-- } \forall a.ProdPar\ i\ a \to a \\
&\textbf{type}\ Producer_{Alt}\ \ o = \forall a.Carrier_R\ o\ a \quad \text{-- } \forall a.ConsPar\ o\ a \to a
\end{aligned}
$$

using their algebras as constructors:

$$
\begin{aligned}
&input_{Alt} :: (i \to Consumer_{Alt}\ i) \to Consumer_{Alt}\ i \\
&input_{Alt} = alg_L \\
&output_{Alt} :: o \to Producer_{Alt}\ o \to Producer_{Alt}\ o \\
&output_{Alt} = alg_R
\end{aligned}
$$

For instance, after fold/build fusion *prodFrom* becomes:

```
prodFrom_Alt :: Integer → Producer_Alt Integer
prodFrom_Alt n = output_Alt n (prodFrom_Alt (n + 1))
```

The merge function for the alternate representations $Producer_{Alt}$ and $Consumer_{Alt}$ then becomes an almost trivial operation.

```
merge_Alt :: Producer_Alt b → Consumer_Alt b → a
merge_Alt p q = q (ProdPar p)
```

3.5 A Not So Special Representation

This merge-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 is included in the appendix.

```
ml⁻¹ :: Consumer_Alt i → Consumer i
ml⁻¹ f = f (ProdPar h) where
  h :: ConsPar i (Consumer i) → Consumer i
  h (ConsPar f) = Consumer (λx → f x (ProdPar h))
mr⁻¹ :: Producer_Alt o → Producer o
mr⁻¹ f = f (ConsPar (λx p → Producer x (h p))) where
  h :: ProdPar o (Producer o) → Producer o
  h (ProdPar f) = f (ConsPar (λx p → Producer x (h p)))
```

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

```
fold_P_Alt :: (o → a → a) → Producer_Alt o → a
fold_P_Alt alg rep = fold_P alg (mr⁻¹ rep)
fold_C_Alt :: ((i → a) → a) → Consumer_Alt i → a
fold_C_Alt alg rep = fold_C alg (ml⁻¹ rep)
```

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

```
fold_P_Alt alg rep = rep (ConsPar (λx p → alg x (h p))) where
  h (ProdPar f) = f (ConsPar (λx p → alg x (h p)))
fold_C_Alt alg rep = rep (ProdPar h) where
  h (ConsPar f) = alg (λx → f x (ProdPar h))
```

4 Return to Two-Sided Pipes

The previous section has derived an efficient approach for 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*.

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

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

```
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 merge 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 Consumer_Alt i = ∀a.ProdPar i a → a
type Producer_Alt 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 Result_R 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_Result_R :: (i → Result_R i o) → Result_R i o
input_Result_R f = λcons (ProdPar prod) →
  prod (ConsPar (λi prod' → f i cons prod'))

output_Result_R :: o → Result_R i o → Result_R i o
output_Result_R o result = λ(ConsPar cons) prod →
  cons o (ProdPar (λcons' → result cons' prod))
```

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

Finally, we can use the one-sided merge function from the previous section to merge 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 merge 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 merge of the two pipes yields a $Result_R\ i\ o$ pipe.

```
merge_Result_R :: Result_R i m → Result_R m o → Result_R i o
merge_Result_R p q = λcons_o prod_i →
  let q' = q cons_o
      p' = flip p prod_i
  in q' (ProdPar p')
```

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
  return x = Cont (λk → k x)
  p >>= f  = Cont (λk → runCont p (λx → runCont (f x) k))
```

If we specialize the result type r to $Result_R\ i\ o$, we get:

```
newtype ContP i o a = ContP ((a → Result_R i o) → Result_R i o)
```

The merge function for *ContP* is implemented in terms of $merge_{Result_R}$.

```
merge_Cont :: ContP i m Void → ContP m o Void → ContP i o a
merge_Cont (ContP p) (ContP q) = ContP (λk →
  merge_Result_R (p absurd) (q absurd))
```

However, there is an issue: before $merge_{Result_R}$ can merge 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 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 don't have a continuation for them, as their continuation is never reached anyway. In short, $merge_{Cont}$ 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 $Result_R\ i\ o$ to *IO* (), which yields:

```
type Result i o = InCont i → OutCont o → IO ()
```

We can rewrite this type as a monad transformer stack, using two reader monad transformers for the two parameters.

```
newtype ReaderT r m a = ReaderT {runReaderT :: r → m a}
type Result' i o = ReaderT (InCont i) (ReaderT (OutCont o) IO) ()
```

Similarly, *ContPipe* $i\ o\ a$ can be written with a transformer stack by adding a *ContT* layer, since *Cont* $(m\ r)$ is equal to *ContT* $r\ m$ for any monad m.

```
newtype ContT r m a = ContT {runContT :: (a → m r) → m r}
type ContPipe' i o a =
  ContT () (ReaderT (InCont i) (ReaderT (OutCont o) IO)) a
```

This transformer stack view enables two additional useful operations: aborting the pipe and embedding an *IO* action. Both are specializations of generic functionality from the continuation monad transformer: *abort* and $lift_{ContT}$.

```
abort :: m r → ContT r m a
abort r = ContT (λk → r)
lift_ContT :: Monad m ⇒ m a → ContT r m a
lift_ContT p = ContT (λk → p ≫= k)
exit' :: ContPipe' i o a
exit' = abort (lift_ReaderT (lift_ReaderT (return ())))
effect' :: IO a → ContPipe' i o a
effect' e = lift_ContT (lift_ReaderT (lift_ReaderT e))
```

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.

The *Proxy* data type at the core of the `pipes` library [**?**] implements bidirectional pipes. The operations *request* and *respond* are respectively downstream and upstream combinations of *input* and *output*. In addition, *Proxy* is also a monad transformer that embed effects of monad 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
```

We refer to the `pipes` source code [**?**] for the implementation of the corresponding $merge_{PL}$ and $merge_{PR}$ functions, which are called +≫ and ≫∼.

We obtain a more efficient function-based representation by adapting the derivation of Sections **??** and **??**. This yields the parameter type *PCPar*.

```
newtype PCPar i o a = PCPar (o → PCPar o i a → a)
```

The $Result_R$ counterpart for *Proxy* takes two such *PCPar*s as input. In addition, the result type r is now a monadic type $m\ r$ to be able to lift operations once it is wrapped with *Cont*.

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

Then, we can proceed with defining the merge function for *ProxyRep* and the *Cont*-wrapped version similar to $Result_R$.

```
merge_ProxyRep :: (c' → ProxyRep a' a c' c m) → ProxyRep c' c b' b m →
  ProxyRep a' a b' b m
merge_ProxyRep fc' q = λreq res →
  let p' c' = fc' c' req
      q' = flip q res
  in q' (PCPar p')
newtype ContPr a' a b' b m r = ContPr {unContPr ::
  (r → ProxyRep a' a b' b m) → ProxyRep a' a b' b m}
merge_ContPr :: (c' → ContPr a' a c' c m Void) →
  ContPr c' c b' b m Void → ContPr a' a b' b m r
merge_ContPr fc' (ContPr q) = ContPr (λk →
  merge_ProxyRep (λc' → unContPr (fc' c') absurd) (q absurd))
```

6 Benchmarks

Figure **??** shows the results of Spivey's *primes* benchmark, which calculates the first n primes. The benchmarks are executed using the `criterion` library [**?**] on an Intel Core i7-6600U at 2.60 GHz with 8 GB memory running Ubuntu 16.04 and GHC 8.4.3, with `-O2` enabled.[2]

The figure compares the `pipes` (v4.3.9) and `conduit` (v1.3.0.3) libraries to Spivey's original implementation (`contpipe`) and our generalized form (`proxyrep`).

We can see that the former two libraries, which use an ADT representation, both show the quadratic performance behaviour for a use case with a high amount of merge steps. On the other hand, the latter two show the improved performance behaviour. The slight overhead of `proxyrep` compared to `contpipe` can be explained by the specialization to *IO* () in the latter type.

The appendix contains the results of some additional microbenchmarks.

7 Related Work

We have covered the main related works of Spivey [**?**], Shivers and Might [**?**] and the `pipes` library [**?**] in the body of the paper. Below we discuss some additional related work.

Encodings The Church [**?**,**?**] and Scott [**?**] 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*.

```
newtype ScottP o = ScottP (∀a.(o → ScottP o → a) → a)
newtype ScottC i = ScottC (∀a.((i → ScottC i) → a) → a)
```

[2] The benchmarks are at `https://github.com/rubenpieters/orth-pipes-bench`.

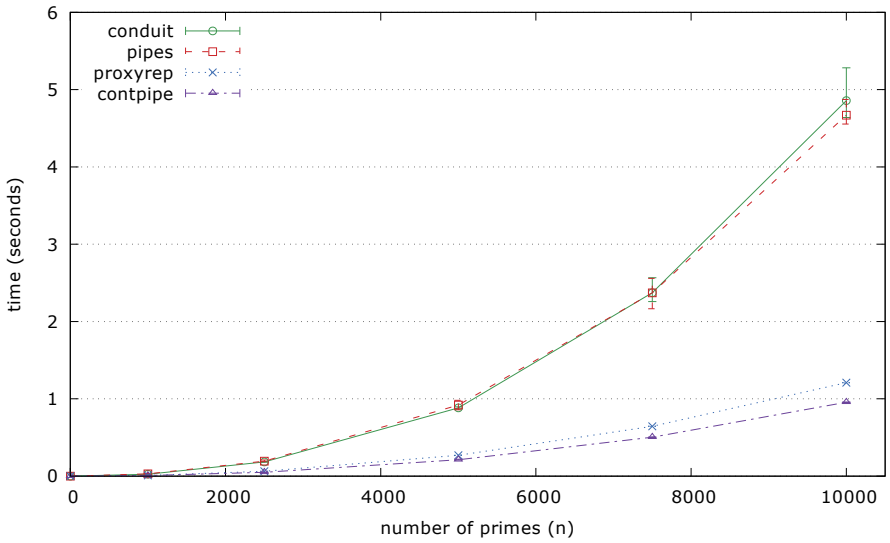

Fig. 1. Results of the *primes* benchmark.

$$\textbf{\textit{newtype}}\ \mathit{SP}\ o\ a = \mathit{SP}\ ((o \to \mathit{SP}\ o\ a \to a) \to a)$$
$$\textbf{\textit{newtype}}\ \mathit{SC}\ i\ a = \mathit{SC}\ (((i \to \mathit{SC}\ i\ a) \to a) \to a)$$

Then, $\forall a.\mathit{SP}\ o\ a$ is representationally equivalent to $\mathit{Producer}_{Alt}$ and similarly for $\forall a.\mathit{SC}\ i\ a$ and $\mathit{Consumer}_{Alt}$ (see the appendix).

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

$$\textbf{\textit{newtype}}\ \mathit{ScottPipe}_\infty\ i\ o = \mathit{ScottPipe}_\infty$$
$$(\forall a.(o \to \mathit{ScottPipe}_\infty\ i\ o \to a) \to ((i \to \mathit{ScottPipe}_\infty\ i\ o) \to a) \to a)$$
$$\textbf{\textit{type}}\ \mathit{SP}_\infty\ i\ o = \forall a.(o \to \mathit{SP}\ o\ a \to a) \to ((i \to \mathit{SC}\ i\ a) \to a) \to a$$

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

Conduit The `conduit` library [**?**] is another popular choice for Haskell stream processing. The two main differing points of `conduit` with `pipes` is 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 untill 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 will be consumed by the downstream with priority. Detecting upstream finalization is handled by *input* returning *Maybe* values, where *Nothing* represents the finalization of the upstream.

Parsers Spivey mentioned in his work [**?**] that the *ContPipe* approach might be adapted to fit the use case of parallel parsers [**?**]. However, after gaining more insight into *ContPipe*, it does not seem that the merging 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 merged structure, but given a non-trivial interpretation. Namely, an interpretation dependent on the other structure during the recursive merge process.

Shallow To Deep Handlers The handlers framework by Kammar et al. [**?**] supports both shallow handlers, based on case analysis, and deep handlers, based on folds. They cover an example of transforming a producer and consumer merging function from shallow handlers to deep handlers. This example is related to our simplified setting in Section **??**. 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 [**?**] 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.

8 Conclusion

We have given an in-depth explanation of the principles behind the fast merging of the three-continuation approach. We have given a series of steps to derive this fast implementation from the less efficient one.

We apply this pattern to the setting of bidirectional pipes, as in the `pipes` library. This results in a more general version of this representation, but still has the same performance due to its efficient merge implementation.

We apply Spivey's benchmarks [**?**] to check that our generalized encoding retains similar performance. We also include the `pipes` library in the benchmark to compare with a commonly used implementation of bidirectional pipes.

This pipes encoding has been made available as a library on github[3].

9 Acknowledgements

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

[3] `https://github.com/rubenpieters/Orthogonal-Pipes`