On the Duality of Streams

How Can Linear Types Help to Solve the Lazy IO Problem?

Jean-Philippe Bernardy
University of Gothenburg
Gothenburg 41255, Sweden

Josef Svenningsson
Chalmers University of Technology
Gothenburg ???, Sweden

treams, Continuations, Linear Types

1 Introduction

As Hughes [12] famously noted, the strength of functional programming languages resides in the composition mechanisms that they provide. That is, simple components can be built and understood in isolation; one does not need to worry about interference effects when composing them. In particular, lazy evaluation affords to construct complex programs by pipelining simple transformation functions. Indeed, whereas strict evaluation forces to fully reify each intermediate result between each computational step, lazy evaluation allows to run all the computations concurrently, often without ever allocating more than a single intermediate element at a time.

Unfortunately, lazy evaluation suffers from two drawbacks. First, it has unpredictable memory behavior. Consider the following function composition:

$$f :: [\,a\,] \to [\,b\,]$$
$$g :: [\,b\,] \to [\,c\,]$$
$$h = g \circ f$$

One hopes that, at run-time, the intermediate list ($[b]$) will only be allocated element-wise, as outlined above. Unfortunately, this desired behavior does not always happen. Indeed, a necessary condition is that the production pattern of f matches the consumption pattern of g; otherwise buffering occurs. In practice, this means that a seemingly innocuous change in either of the function definitions may drastically change the memory behavior of the composition, without warning. If one cares about memory behavior, this means that the compositionality principle touted by Hughes breaks down.

Second, lazy evaluation does not extend nicely to effectful processing. That is, if (say) an input list is produced by reading a file lazily, one is exposed to losing referential transparency (as Kiselyov [17] has shown). For example, one may rightfully expect[1] that both following programs have the same behavior:

$$main = \textbf{do}\ inFile \leftarrow openFile\ \texttt{"foo"}\ ReadMode$$
$$contents \leftarrow hGetContents\ inFile$$
$$putStr\ contents$$
$$hClose\ inFile$$

$$main = \textbf{do}\ inFile \leftarrow openFile\ \texttt{"foo"}\ ReadMode$$
$$contents \leftarrow hGetContents\ inFile$$
$$hClose\ inFile$$
$$putStr\ contents$$

Indeed, the *putStr* and *hClose* commands act on unrelated resources, and thus swapping them should have no observable effect. However, while the first program prints the `foo` file, the second one prints nothing. Indeed, because *hGetContents* reads the file lazily, the *hClose* operation has the effect to truncate the list. In the first program, printing the contents forces reading the file. One may argue that *hClose* should not be called in the first place — but then, closing the handle happens only when the *contents* list can be garbage collected (in full), and relying on garbage collection for cleaning resources is brittle; furthermore this effect compounds badly with the first issue discussed above (unpredictability of buffering). If one wants to use lazy effectful computations, again, the compositionality principle is lost.

In this paper, we propose to tackle both of these issues by taking advantage of linear types. One way to read this paper is as an advocacy for linear types support in Haskell. Indeed, even though one can tackle the above issues by using Monad-based libraries [16] our linear-logic-based types naturally capture a wealth of useful production and consumption patterns.

The first noteworthy type, corresponding to on-demand production of elements, is called a source (*Src*). An adaptation of the first code example above to use sources would look as follows.

$$f :: Src\ a \multimap Src\ b$$
$$g :: Src\ b \multimap Src\ c$$
$$h = g \circ f$$

Thanks to type-checking, we get the guarantee that the composition does not allocate more memory than the sum of its components. The second useful type, driving the consumption of elements, is called a sink (*Snk*). For example, the standard output is naturally given a sink type:

$$stdoutSnk :: Snk\ String$$

---

[1] This expectation is expressed in a Stack Overflow question, accessible at this URL: http://stackoverflow.com/questions/296792/haskell-io-and-closing-files

Using it, we can implement the printing of a file as follows.

$main = fileSrc\ \texttt{"foo"}\ `fwd`\ stdoutSnk$

Beyond frugal memory usage, we have the guarantee of a timely release of resources, even in the presence of exceptions. In the above, *fileSrc* provides the contents of a file, and *fwd* forwards data from a source to a sink. The types are as follows:

$fileSrc :: FilePath \rightarrow Src\ String$

$fwd :: Src\ a \multimap Snk\ a \multimap IO\ ()$

Sources provide data on-demand, while sinks decide when they are ready to consume data. This is an instance of the push/pull duality. In general, push-streams control the flow of computation, while pull-streams respond to it. We will see that this polarization does not need to match the flow of data: for example we support data sources with push-flavor, called co-sources (*CoSrc*). Co-sources are useful for example when a source data stream needs precise control over the execution of effects it embeds (see Sec. 6). For example, sources cannot be demultiplexed, but co-sources can.

In a program which uses both sources and co-sources, the need might arise to compose a function which returns a co-source with a function which takes a source as input: this is the situation where list-based programs would silently cause memory allocation. In our approach, this mismatch is caught by the type system, and the user must explicitly conjure a buffer to be able to write the composition:

$f :: Src\ a \multimap CoSrc\ b$

$g :: Src\ b \multimap Src\ c$

$h = g \circ buffer \circ f$

The contributions of this paper are

- The formulation of principles for compositional resource-aware programming in Haskell (resources include memory and files), based on linearity and polarization. While these principles are borrowed from linear logic, as far as we know they have not been applied to Haskell programming before.
- An embodiment of the above principles, in the form of a Hask-LL[2] library for streaming IO. Besides supporting compositionality as outlined above, our library features two concrete novel aspects:
  1. A more lightweight design than state-of-the-art co-routine based libraries.
  2. Support for explicit buffering and control structures, while still respecting compositionality (Sec. 6).

Outline The rest of the paper is structured as follows. In Sec. 2, we recall the notions of continuations in presence of effects. In Sec. 3, we present our design for streams, and justify it by appealing to linearity principles. In Sec. 4, we give an API to program with streams, and analyze their algebraic structure. In Sec. 5, we show how to embed IO into streams. In Sec. 6, we discuss polarity mismatch. Related

work and future work are discussed respectively in sections 7 and 8. We conclude in Sec. 9.

2 Preliminary: negation and continuations

In this section we recall the basics of continuation-based programming. We introduce our notation, and justify effectful continuations.

We begin by assuming a type of effects *Eff*, which we keep abstract for now. We can then define negation as follows:

type $N\ a = a \multimap Eff$

A shortcut for double negations is also convenient.

type $NN\ a = N\ (N\ a)$

The basic idea (imported from classical logic) pervading this paper is that in the presence of effects, producing a result of type α is equivalent to consuming an argument of type $N\alpha$. Dually, consuming an argument of type α is equivalent to producing a result of type $N\alpha$. We call these equivalences the duality principle.

In classical logic, negation is involutive; that is: $NN\ \alpha = \alpha$. However, because we work within Haskell, we do not have this equality[3]. We can come close enough though. First, double negations can always be introduced, using the *shift* operator:

$shift :: a \multimap NN\ a$

$shift\ x\ k = k\ x$

Second, it is possible to remove double negations, but only if an effect can be outputted. Equivalently, triple negations can be collapsed to a single one:

$unshift :: N\ (NN\ a) \multimap N\ a$

$unshift\ k\ x = k\ (shift\ x)$

Aside. The above two functions are the *return* and *join* of the double negation monad[4]; indeed adding a double negation in the type corresponds to sending the return value to its consumer. However, we will not be using this monadic structure anywhere in the following, because single negations play a central role in our approach. The monadic structure is a mere diversion.

2.1 Structure of Effects

When dealing with purely functional programs, continuations have no effects. In this case, one can let *Eff* remain abstract, or define it to be the empty type: $Eff = \bot$. This choice is also natural when interpreting the original linear logic of Girard [10], Laurent [21].

The pure logic treats effects purely abstractly, but interpretations may choose to impose a richer structure on them. Such interpretations would then not be complete with respect to the logic — but they would remain sound. In our case, we first require *Eff* to be a monoid. Its unit (*mempty*) corresponds to program termination, while the operator (*mappend*) corresponds to sequential composition

---

[2]Hask-LL is an extension of Haskell to linear types fully described by Bernardy et al. [4]. For the purposes of this paper, the main addition is the linear arrow ($\multimap$) for functions which guarantee to consume their argument exactly once. Pairs also become linear.

[3]Even though Munch-Maccagnoni [23] achieves an involutive negation in an intuitionistic language, he does so by stack manipulation, which is not available in Haskell.

[4]for *join*, substitute $N\ a$ for a in the type of *unshift*

of effects. (This structure is standard to interpret the HALT and MIX rules in linear logic [3, 22])

For users of the stream library, *Eff* shall remain an abstract monoid. However in this paper we develop concrete effectful streams, and therefore we greatly extend the structure of effects. In fact, because we will provide streams interacting with files and other operating-system resources, and write the whole code in standard Hask-LL, we must pick $Eff = IO()$, and ensure that *Eff* can be treated as a linear monoid.

type $Eff = IO$ ()

class *Monoid a* **where**
  $mempty :: a$
  $(\diamond) :: a \multimap a \multimap a$
instance *Monoid Eff*

The parts of the code which know about $Eff = IO()$ must be carefully written, because the type system does not provide strong guarantees about such code. In fact these IO-interacting functions do not interpret $\perp$ in a standard way: they are non-standard extensions of its model.

3 Streams

Our main guiding design principle is duality. In fact we will make crucial use of duality in the later sections of this paper. The duality principle is reflected in the design of the streaming library: we not only have a type for sources of data but also a type for sinks. For example, a simple stream processor reading from a single source and writing to a single sink may be given the following type:

$simple :: Src\ a \multimap Snk\ a \multimap Eff$

We will make sure that *Snk* is the negation of a source (and vice versa), and thus the type of the above program may equivalently be given the following type:

$simple :: Src\ a \multimap Src\ a$

However, having explicit access to sinks allows us to (for example) dispatch a single source to multiple sinks, as in the following type signature:

$forkSrc :: Src\ (a, b) \multimap Snk\ a \multimap Snk\ b \multimap Eff$

We define sources and sinks by mutual recursion. Producing a source means to select if some more is available (*Cons*) or not (*Nil*). If there is data, one must then produce a data item and *consume* a sink.

data *Source a* **where**
  $Nil :: Source\ a$
  $Cons :: a \multimap N\ (Sink\ a) \multimap Source\ a$
data *Sink   a* **where**
  $Full :: Sink\ a$
  $Cont :: (N\ (Source\ a)) \multimap Sink\ a$

Producing a sink means to select whether one can accept more elements (*Cont*) or not (*Full*). In the former case, one must then be able to consume a source. The *Full* case is useful when the sink bails out early, for example when it runs out of space.

Note that, in order to produce (or consume) the next element, the source (or sink) must handle the effects generated by the other side of the stream before proceeding. This means that each production is matched by a consumption, and *vice versa*; element-wise.

3.1 Linearity

For streams to be used safely, one cannot discard nor duplicate them, for otherwise effects may be discarded and duplicated, which is dangerous. For example, the same file could be closed twice, or not at all. Indeed, the last action of a sink typically involves closing the file. Timely closing of the sink can only be guaranteed if the actions are run until reaching the end of the pipe (either *Full* or *Nil*).

Linear types allow to capture such an invariant: all functions from our library treat sources and sinks linearly. We underline that even individual elements are declared linear, which allow us to store even effects in them.

3.2 Basics

As a first illustration, we present three basic functions to manipulate *Source*s and *Sink*s: one to read from sources, one to write to sinks, and one to connect sources and sinks.

Reading One may attempt to write the following function, waiting for data to be produced by a source. The second argument is the effect to run if no data is produced, and the third is the effect to run given the data and the remaining source.

$await :: Source\ a \multimap Eff \multimap (a \multimap Source\ a \multimap Eff) \multimap Eff$
$await\ Nil\qquad\quad eof\ \_ = eof$
$await\ (Cons\ x\ cs)\ \_\quad k = cs\ \$\ Cont\ \$\ \lambda xs \to k\ x\ xs$

However, the above function breaks linearity (*eof* and *k* are not always used), so we cannot define it as such. Instead we have to arrange the types so that *await* can choose *itself* between the *eof* continuation and *k*. To do so, we must provide them as a so-called additive conjunction. The additive conjunction is the dual of the *Either* type: there is a choice, but this choice falls on the consumer rather than the producer of the input. The additive conjunction, written &, can be encoded by sandwiching *Either* between two inversions of the control flow, thereby switching the party which makes the choice:

type $a\ \&\ b = N\ (Either\ (N\ a)\ (N\ b))$

(One will recognize the similarity between this definition and the De Morgan's laws.) Await can then be written as follows:

$await :: Source\ a \multimap (Eff\ \&\ (a \multimap Source\ a \multimap Eff)) \multimap Eff$
$await\ Nil\ r = r\ \$\ Left\ \$\ \lambda eof \to eof$
$await\ (Cons\ x\ xs)\ r = r\ \$\ Right\ \$\ \lambda c \to xs\ (Cont\ (c\ x))$

Writing One can also write data into a sink, and obtain a new, (doubly negated) sink which represents the state of the sink after the "old" sink has consumed the data. If the sink is full, the data is ignored.

$yield :: a \to Sink\ a \multimap (Sink\ a \multimap Eff) \multimap Eff$
$yield\ x\ (Cont\ c)\ k = c\ (Cons\ x\ k)$
$yield\ x\ Full\ k = k\ Full$

3.3 Baking in negations

Programming with *Source* and *Sink* explicitly is inherently continuation-heavy. As we have seen in the definition of `unshift`, double-negations cannot simply be removed: effects must be collected and thus most functions must return an `Eff`. Thus it is convenient to introduce negated variants of source and sink types.

type $Src\ a = N\ (Sink\ a)$
type $Snk\ a = N\ (Src\ a)$

These definitions have the added advantage to perfect the duality between sources and sinks, while not restricting the programs one can write. A benefit of the above definitions is that it becomes possible to forward data from *Src* to *Snk* without dropping elements:

$fwd :: Src\ a \multimap Snk\ a \multimap Eff$
$fwd = shift$

We additionally introduce the following type synonym, which is used to write the types of functions which manipulate `Source`s directly.

type $Snk'\ a = N\ (Source\ a)$

Functions over Snk' can be lifted manipulate *Src*, and in turn *Snk*. We can do so generically for endomorphisms:

$flipSnk :: (Snk'\ a \multimap Snk'\ b) \to Src\ b \multimap Src\ a$
$flipSrc :: (Src\ a\quad \multimap Src\ b) \to Snk\ b \multimap Snk\ a$
$flipSnk\ \_\ s\ Full = s\ Full$
$flipSnk\ f\ s\ (Cont\ k) = s\ \$\ Cont\ \$\ (f\ k)$
$flipSrc\ f\ snk\ src = snk\ (f\ src)$

As a particular case, consider the mapping functions:

$mapSrc\ :: (a \multimap b) \to Src\ \ a \multimap Src\ \ b$
$mapSnk :: (b \multimap a) \to Snk\ a \multimap Snk\ b$

Mapping sinks is defined by flipping the mapping of sources; and mapping of sources is defined by flipping mapping of negated sources:

$mapSnk\ f = flipSrc\ (mapSrc\ f)$
$mapSrc\ f = flipSnk\ (mapSnk'\ f)$

Mapping of negated sources is defined by case analysis on the concrete source, and the recursive case conveniently calls *mapSrc*.

$mapSnk'\ \_\ snk\ Nil = snk\ Nil$
$mapSnk'\ f\ snk\ (Cons\ a\ s)$
$\quad = snk\ (Cons\ (f\ a)\ (mapSrc\ f\ s))$

When using double negations, it is sometimes useful to insert or remove them inside type constructor. Introduction of double negation in sources and its elimination in sinks is a special case of mapping.

$nnIntro :: Src\ a \multimap Src\ (NN\ a)$
$nnIntro = mapSrc\ shift$

$nnElim' :: Snk\ (NN\ a) \multimap Snk\ a$
$nnElim' = mapSnk\ shift$

The duals are implemented by case analysis, following the mutual recursion pattern introduced above.

$nnElim :: Src\ (NN\ a) \multimap Src\ a$
$nnIntro' :: Snk'\ a \multimap Snk'\ (NN\ a)$

$nnElim = flipSnk\ nnIntro'$
$nnIntro'\ k\ Nil = k\ Nil$
$nnIntro'\ k\ (Cons\ x\ xs) = x\ \$\ \lambda x' \to k\ (Cons\ x'\ \$\ nnElim\ xs)$

4 Effect-Free Streams

The functions seen so far make no use of the fact that *Eff* can embed IO actions. Indeed, a large number of useful functions over streams can be implemented without relying on IO. We give an overview of effect-free streams in this section.

4.1 List-Like API

To begin, we show that one can implement a list-like API for sources, as follows:

$empty :: Src\ a$
$empty\ Full = mempty$
$empty\ (Cont\ k) = k\ Nil$

$cons :: a \multimap Src\ a \multimap Src\ a$
$cons\ a\ s\ s' = yield\ a\ s'\ s$

(Taking the head or tail is not meaningful due to the linearity constraints: *await* must be used instead.)

Another useful function is the equivalent of *take* on lists. Given a source, we can create a new source which ignores all but its first n elements. Conversely, we can prune a sink to consume only the first n elements of a source. We can even do it without breaking linearity, because all streams support early stopping.

$takeSrc :: Int \to Src\ \ a \multimap Src\ \ a$
$takeSnk :: Int \to Snk\ a \multimap Snk\ a$

The natural implementation is again by mutual recursion. The main subtlety is that, when reaching the nth element, both ends of the stream must be notified of its closing. Note the use of the monoidal structure of *Eff* in this case.

$takeSrc\ 0\ s\ t = s\ Full \diamond empty\ t$
$takeSrc\ i\ s\ t = flipSnk\ (takeSnk'\ i)\ s\ t$

$takeSnk'\ \_\ s\ Nil = s\ Nil$
$takeSnk'\ i\ s\ (Cons\ a\ s') = s\ (Cons\ a\ (takeSrc\ (i-1)\ s'))$

$takeSnk\ n = flipSrc\ (takeSrc\ n)$

4.2 Algebraic structure

Streams form (linear) monoids under concatenation:

instance $Monoid\ (Src\ a)$ **where**
$\quad (\diamond) = appendSrc$
$\quad mempty = empty$

instance $Monoid\ (Snk\ a)$ **where**
$\quad (\diamond) = appendSnk$
$\quad mempty = shift\ Full$

We have already encountered *empty* above; the appending operations are defined below. Intuitively, *appendSrc* first gives

control to the first source until it runs out of elements and then turns control over to the second source. This behavior is implemented in the helper function *forwardThenSnk*.

$$appendSrc :: Src\ a \multimap Src\ a \multimap Src\ a$$
$$appendSrc\ s1\ s2\ Full = s1\ Full \Diamond s2\ Full$$
$$appendSrc\ s1\ s2\ (Cont\ s)$$
$$\quad = s1\ (Cont\ (forwardThenSnk'\ s\ s2))$$

$$forwardThenSnk' :: Snk'\ a \multimap Src\ a \multimap Snk'\ a$$
$$forwardThenSnk'\ snk'\ src\ Nil = src\ (Cont\ snk')$$
$$forwardThenSnk'\ snk'\ src\ (Cons\ a\ s)$$
$$\quad = snk'\ (Cons\ a\ (appendSrc\ s\ src))$$

Sinks can be appended is a similar fashion.

$$appendSnk' :: Snk'\ a \multimap Snk'\ a \multimap Snk'\ a$$
$$appendSnk'\ s1\ s2\ Nil = s1\ Nil \Diamond s2\ Nil$$
$$appendSnk'\ s1\ s2\ (Cons\ a\ s)$$
$$\quad = s1\ (Cons\ a\ (forwardThenSrc\ s2\ s))$$

$$forwardThenSrc :: Snk'\ a \multimap Src\ a \multimap Src\ a$$
$$forwardThenSrc\ s2 = flipSnk\ (appendSnk'\ s2)$$

$$appendSnk :: Snk\ a \to Snk\ a \to Snk\ a$$
$$appendSnk\ t1\ t2\ s = t1\ \$\ \lambda\textbf{case}$$
$$\quad Full \to t2\ empty \Diamond s\ Full$$
$$\quad Cont\ k \to flipSrc\ (forwardThenSrc\ k)\ t2\ s$$

The operations *forwardThenSnk* and *forwardThenSrc* are akin to taking the difference between a source and a sink. Thus we find it convenient to give them the following aliases:

$$(-?) :: Snk'\ a \multimap Src\ a \multimap Snk'\ a$$
$$t -?\ s = forwardThenSnk'\ t\ s$$

$$(-!) :: Snk'\ a \multimap Src\ a \multimap Src\ a$$
$$t -!\ s = forwardThenSrc\ t\ s$$

infixr $-!$
infixl $-?$

Appending and subtracting interact in the expected way. That is, the following equalities hold observationally:

$$t -?\ (s1 \Diamond s2) \equiv t -?\ s2 -?\ s1$$
$$(t1 \Diamond t2) -!\ s \equiv t1 -!\ t2 -!\ s$$

Functor We have already seen the mapping functions for sources and sinks: sources are functors and sinks are contravariant functors. (Given the implementation of the morphism actions it is straightforward to check the functor laws.)

4.3 Table of effect-free functions

The above gives already an extensive API for sources and sinks, many more useful effect-free functions can be implemented on this basis. We give here a menu of functions that we have implemented, and whose implementation is available online[5].

Two sources can be zipped if some extra left-over elements can be dropped. Equivalently, a sink can be forked in the same situation.

---

[5]https://github.com/jyp/organ/blob/master/Organ-HaskeLL.lhs

class *Drop a* **where** $drop :: a \multimap b \multimap b$

$$zipSrc :: (Drop\ a, Drop\ b) \Rightarrow Src\ a \multimap Src\ b \multimap Src\ (a, b)$$
$$forkSnk :: (Drop\ a, Drop\ b) \Rightarrow Snk\ (a, b) \multimap Src\ a \multimap Snk\ b$$

Zip two sinks, and the dual.

$$forkSrc :: Src\ (a, b) \multimap Snk\ a \multimap Src\ b$$
$$zipSnk :: Snk\ a \multimap Snk\ b \multimap Snk\ (a, b)$$

Equivalent of *scanl'* for sources, and the dual

$$scanSrc :: (b \to a \multimap (b, c)) \to b \to Src\ a \multimap Src\ c$$
$$scanSnk :: (b \to a \multimap (b, c)) \to b \to Snk\ c \multimap Snk\ a$$

Equivalent of *foldl'* for sources, and the dual.

$$foldSrc' :: (b \to a \multimap b) \to b \to Src\ a \multimap NN\ b$$
$$foldSnk' :: (b \to a \multimap b) \to b \to N\ b \multimap Snk\ a$$

Drop some elements from a source, and the dual.

$$dropSrc :: Drop\ a \Rightarrow Int \to Src\ a \multimap Src\ a$$
$$dropSnk :: Drop\ a \Rightarrow Int \to Snk\ a \multimap Snk\ a$$

Convert a list to a source, and vice versa.

$$fromList :: [a] \multimap Src\ a$$
$$toList :: Src\ a \multimap NN\ [a]$$

Split a source in lines, and the dual.

$$linesSrc :: Src\ Char \multimap Src\ String$$
$$unlinesSnk :: Snk\ String \multimap Snk\ Char$$

Interleave two sources, and the dual.

$$interleave :: Src\ a \multimap Src\ a \multimap Src\ a$$
$$interleaveSnk :: Snk\ a \multimap Src\ a \multimap Snk\ a$$

Forward data coming from the input source to the result source and to the second argument sink.

$$tee :: (a \multimap (b, c)) \to Src\ a \multimap Snk\ b \multimap Src\ c$$

Filter a source, and the dual.

$$filterSrc :: (a \multimap Maybe\ b) \multimap Src\ a \multimap Src\ b$$
$$filterSnk :: (a \multimap Maybe\ b) \multimap Snk\ b \multimap Snk\ a$$

Turn a source of chunks of data into a single source; and the dual.

$$unchunk :: Src\ [a] \multimap Src\ a$$
$$chunkSnk :: Snk\ a \multimap Snk\ [a]$$

4.4 App: Stream-Based Parsing

To finish with effect-free functions, we give an example of a stream processor which applies a parser to a source of symbols and returns a source of parse results. This conversion is useful for example to turn an XML file, provided as a stream of characters into a stream of (opening and closing) tags.

We begin by defining a pure parsing structure, modeled after the parallel parsing processes of Claessen [6]. The parser is continuation based, but the effects being accumulated are parsing processes, defined as follows. The *Sym* constructor parses *Just* a symbol, or *Nothing* if the end of stream is reached. A process may also *Fail* or return a *Result*.

```
data P s res = Sym (Maybe s → P s res)
             | Fail
             | Result res
```

A parser is producing the double negation of a:

```
newtype Parser s a = P (∀res.(a → P s res) → P s res)
```

The monadic interface can then be built in the standard way:

```
instance Monad (Parser s) where
  return x = P $ λfut → fut x
  P f ≫ k = P (λfut → f (λa → let P g = k a in g fut))
instance Applicative (Parser s) where
  pure = return
  (< ∗ >) = ap
instance Functor (Parser s) where
  fmap = (<$>)
```

The essential parsing ingredient, choice, rests on the ability to weave processes together; picking that which succeeds first, and that which fails as last resort:

```
weave :: P s a → P s a → P s a
weave Fail x = x
weave x Fail = x
weave (Result res) _ = Result res
weave _ (Result res) = Result res
weave (Sym k1) (Sym k2)
    = Sym (λs → weave (k1 s) (k2 s))

(<|>) :: Parser s a → Parser s a → Parser s a
P p <|> P q = P (λfut → weave (p fut) (q fut))
```

Parsing then reconciles the execution of the process with the traversal of the source. In particular, whenever a result is encountered, it is fed to the sink. If the parser fails, both ends of the stream are closed.

```
parse :: ∀s a.Parser s a → Src s ⊸ Src a
parse _ src Full = src Full
parse q@(P p0) src (Cont k) = scan (p0 $ λx → Result x) k src
  where
scan :: P s a → Snk' a ⊸ Src s ⊸ Eff
scan (Result res) ret   xs = ret (Cons res (parse q xs))
scan Fail         ret   xs = ret Nil <> xs Full
scan (Sym f)      mres xs = xs $ Cont $ λcase
  Nil        → scan (f Nothing) mres empty
  Cons x cs → scan (f $ Just x) mres cs
```

5 Effectful streams

So far, we have constructed only effect-free streams. That is, effects could be any monoid, and in particular the unit type. In this section we bridge this gap and provide some useful sources and sinks performing IO effects, namely reading and writing to files.

We first define the following helper function, which sends data to a handle, thereby constructing a sink.

```
hFileSnk' :: Handle → Snk' String
hFileSnk' h Nil = hClose h
hFileSnk' h (Cons c s) = do
```

```
  hPutStrLn h c
  hFileSnk h s
```

```
hFileSnk :: Handle → Snk String
hFileSnk h = shift (Cont (hFileSnk' h))
```

A file sink is then simply:

```
fileSnk :: FilePath → Snk String
fileSnk file s = do
  h ← openFile file WriteMode
  hFileSnk h s
```

And the sink for standard output is (the 1 indicates that there is a single instance of this resource):

```
stdoutSnk :: 1 Snk String
stdoutSnk = hFileSnk stdout
```

(For ease of experimenting with our functions, the data items are lines of text — but an production-strength version would provide chunks of raw binary data, to be further parsed.)

Conversely, a file source reads data from a file, as follows:

```
hFileSrc :: Handle → Src String
hFileSrc h Full = hClose h
hFileSrc h (Cont c) = do
  e ← hIsEOF h
  if e then do hClose h
               c Nil
       else  do x ← hGetLine h
                c (Cons x $ hFileSrc h)
```

```
fileSrc :: FilePath → Src String
fileSrc file sink = do
  h ← openFile file ReadMode
  hFileSrc h sink
```

Combining the above primitives, we can then implement file copy as follows:

```
copyFile :: FilePath → FilePath → Eff
copyFile source target = fwd (fileSrc source)
                             (fileSnk target)
```

It should be emphasized at this point that when running *copyFile* reading and writing will be interleaved: in order to produce the next line in the source (in this case by reading from the file), the current line must first be consumed in the sink (in this case by writing it to disk). The stream behaves fully synchronously, and no intermediate data is buffered.

Whenever a sink is full, the source connected to it will be finalized. The next example shows what happens when a sink closes the stream early. Instead of connecting the source to a bottomless sink, we connect it to one which stops receiving input after three lines.

```
read3Lines :: Eff
read3Lines = fwd (hFileSrc stdin)
                 (takeSnk 3 $ fileSnk "text.txt")
```

Indeed, testing the above program reveals that it properly closes *stdin* after reading three lines. This early closing is critical to modular stream programming: it allows proper

finalization of one end when an exception occurs at the other end.

5.1   Exception Handling

While the above implementations of file source and sink are fine for illustrative purposes, their production-strength versions should handle exceptions. Doing so is straightforward: as shown above, our sinks and sources readily support early closing of the stream.

The following code fragment shows how to handle an exception when reading a line in a file source.

```
hFileSrcSafe :: Handle → Src String
hFileSrcSafe h Full = hClose h
hFileSrcSafe h (Cont c) = do
    e ← hIsEOF h
    if e then do
        hClose h
        c Nil
    else do
        mx ← catch (Just <$> hGetLine h)
                   (λ(_ :: IOException) → return Nothing)
        case mx of
            Nothing → c Nil
            Just x → c (Cons x $ hFileSrcSafe h)
```

Exceptions raised in *hIsEOF* should be handled in the same way. The file sink is responsible for handling its own exceptions so there is no need to insert a handler around the invocation of the continuation *c*. Thus, dealing with exceptions is done once and for all when implementing the library of streams. The programmer using the library does not have to be concerned with exceptions as they are caught and communicated properly under the hood, even if one would probably have a component of the *Eff* type indicating the nature of the exception encountered, if any. Yet we leave it out in the proof of concept implementation presented in this paper.

Using exception handlers, as in the above snippet, will secure the library from synchronous exceptions arising from accessing the file, but not from asynchronous exceptions which may come from other sources. Asynchronous exception-safety requires more machinery. The region library presented in Kiselyov and Shan [18] can be used for this purpose, as outlined in Kiselyov [15].

6   Synchronicity and Asynchronicity

One of the main benefits of streams as defined here is that the details of synchronizing concrete sink and sources are abstracted over. That is, one can build a data source regardless of how the data is be consumed, or dually one can build a sink regardless of how the data is produced; but, despite the independence of definitions, all the code can (and is) executed synchronously: composing a source and a sink require no concurrency (nor any external control structure).

As discussed above, a consequence of synchronicity is that the programmer cannot be implicitly buffering data when connecting a source to a sink: every production must be matched by a consumption (and vice versa). In sum, synchronicity restricts the kind of operations one can construct, in exchange for two guarantees:

1. The effects of connected sources and sinks are run synchronously
2. No implicit memory allocation happens

While the guarantees have been discussed so far, it may be unclear how synchronicity actually restricts the programs one can write. In the rest of the section we show by example how the restriction plays out.

6.1   Example: demultiplexing

One operation supported by synchronous behavior is the demultiplexing of a source, by connecting it to two sinks (and the dual operation of mutiplexing sinks).

```
dmux' :: Src (Either a b) → Sink a → Sink b → Eff
dmux' sab Full tb = sab Full <> empty tb
dmux' sab ta Full = sab Full <> empty ta
dmux' sab (Cont ta) (Cont tb) = sab $ Cont $ λs → dmux ta tb s

dmux :: Snk' a → Snk' b → Snk' (Either a b)
dmux ta tb Nil = ta Nil <> tb Nil
dmux ta tb (Cons ab c) = case ab of
    Left a → ta (Cons a $ λta' → dmux' c ta' (Cont tb))
    Right b → tb (Cons b $ λtb' → dmux' c (Cont ta) tb')
```

The key ingredient is that demultiplexing starts by reading the next value available on the source. Depending on its value, we feed the data to either of the sinks and proceed. Besides, as soon as any of the three parties closes the stream, the other two are notified.

However, multiplexing sources cannot be implemented while respecting synchronicity. To see why, let us attempt anyway, using the following type signature:

```
mux :: Src a ⊸ Src b ⊸ Src (Either a b)
mux sa sb = ?
```

We can try to fill the hole by reading on a source. However, if we do this, it falls onto the multiplexer to choose which source to run first. We may pick *sa*, however it may be blocking, while *sb* is ready with data. This is not really multiplexing, at best this approach would give us interleaving of data sources, by taking turns.

In order to make any progress, we can let the choice of which source to pick fall on the consumer of the stream. The type that we need for output data in this situation is the additive conjunction. We can then amend the type of multiplexing:

```
mux :: Src a ⊸ Src b ⊸ Src (a & b)
```

Unfortunately, we still cannot implement multiplexing with the above type. Indeed, consider the following attempt, where we begin by asking the consumer if it desires *a* or *b*. If the answer is *a*, we can extract a value from *sa* and yield it; and symmetrically for *b*.

```
mux sa sb (Cont tab) = tab $ Cons
    (λab → case ab of
        Left  ka → sa $ Cont $ λ(Cons a resta) → ka a
```

$Right\ kb \to sb\ \$\ Cont\ \$\ \lambda(Cons\ b\ restb) \to kb\ b)$
$(error\ \texttt{"oops"})$

However, there is no way to then make a recursive call (oops) to continue processing. Indeed the recursive call to make must depend on the choice made by the consumer (in one case we should be using *resta*, in the other *restb*). However the type of *Cons* forces us to produce its arguments independently.

What we need to do is to reverse the control fully: we need a data source which is in control of the flow of execution.

6.2 Co-Sources, Co-Sinks

We call the structure that we are looking for a *co-source*. Co-sources are the subject of this section. We remember that producing Na is equivalent to consuming a, thus that a sink of Na is a (different kind of) source of a. We define:

type $CoSrc\ a = Snk\ (N\ a)$
type $CoSnk\ a = Src\ (N\ a)$

Implementing multiplexing on co-sources is then straightforward, by leveraging $dmux'$:

$mux' :: CoSrc\ a \multimap CoSrc\ b \multimap CoSrc\ (a\ \&\ b)$
$mux'\ sa\ sb\ tab = sa\ \$\ \lambda sa' \to sb\ \$\ \lambda sb' \to dmux'\ (nnElim\ tab)\ sa'\ sb'$

We use the rest of this section to study the property of co-sources and co-sinks. Firstly *CoSrc* is a functor, and *CoSnk* is a contravariant functor.

$mapCoSnk :: (b \multimap a) \to CoSnk\ a \multimap CoSnk\ b$
$mapCoSnk\ f = mapSrc\ (\lambda b' \to \lambda a \to b'\ (f\ a))$

$mapCoSrc :: (a \multimap b) \to CoSrc\ a \multimap CoSrc\ b$
$mapCoSrc\ f\ snk\ src = snk\ (mapCoSnk\ f\ src)$

Elements of a co-source are accessed only in when and in the order that the co-source imposes. Consequently, one cannot in general extract the contents of a co-source as a list. Attempting to implement this extraction looks as follows.

$coToList :: Snk'\ (N\ a) \to NN\ [a]$
$coToList\ k1\ k2 = k1\ \$\ Cons\ (\lambda a \to k2\ [a])\ (error\ \texttt{"rest"})$
$coToList\ k1\ k2 = k2\ \$\ (error\ \texttt{"a?"}) : (error\ \texttt{"rest"})$

If one tries to begin by eliminating the co-source (first equation), then there is no way to produce subsequent elements of the list. If one tries to begin by constructing the list (second equation), then no data is available.

Yet it is possible to define useful co-sources and co-sinks. The first example shows how to provide a file as a co-source:

$coFileSrc :: Handle \to CoSrc\ String$
$coFileSrc\ h\ snk = \textbf{do}$
$\quad e \leftarrow hIsEOF\ h$
$\quad snk\ \$\ \textbf{if}\ \neg\ e$
$\quad\quad \textbf{then}\ Full$
$\quad\quad \textbf{else}\ Cont\ \$\ \lambda\textbf{case}$
$\quad\quad\quad Nil \to hClose\ h$
$\quad\quad\quad Cons\ x\ xs \to \textbf{do}$
$\quad\quad\quad\quad x' \leftarrow hGetLine\ h$
$\quad\quad\quad\quad x\ x' \qquad\qquad \text{-- (1)}$
$\quad\quad\quad\quad (coFileSrc\ h)\ xs \quad \text{-- (2)}$

Compared to *fileSrc*, the difference are that this function decides the ordering of effects ran in a co-sink connected to it. That is, even though there is no data dependency between the lines (1) and (2), they are run in a specific, given order. We will see in the next section how this situation generalizes.

The second example is an infinite co-sink that sends data to a file.

$coFileSink :: Handle \to CoSnk\ String$
$coFileSink\ h\ Full = hClose\ h$
$coFileSink\ h\ (Cont\ c) = c\ (Cons\ (hPutStrLn\ h)$
$\qquad\qquad\qquad\qquad\qquad (coFileSink\ h))$

Compared to *fileSnk*, the difference is that the program does not control the order of execution of effects. The effect of writing the current line is put in a data structure, and its execution is up to the co-source which eventually connects to the co-sink. Thus, the order of writing lines in the file depends on the order of effects chosen in the co-source connected to this co-sink.

In sum, shifting from streams to co-streams shifts the control flow responsibility from the sink to the (co-)source. It should be stressed that, in the programs which use the functions defined so far (even those that use *IO*), synchronicity is preserved: no data is buffered implicitly, and reading and writing occur in lockstep.

6.3 Asynchronicity

We have seen so far that synchronicity gives useful guarantees, but restricts the kind of programs one can write. In this section, we provide primitives which allow controlled asynchronous programming within our framework. The main benefit of sticking to our framework in this case is that asynchronous behavior is circumscribed to the explicit usages of these primitives. That is, the benefits of synchronous programming still hold locally.

Scheduling Asynchronicity arises either from an excess or a deficit of control. Let us first examine the latter case, which involves two streams that do not control the flow of execution. This means that effects must be scheduled explicitly and externally to the streams. This situation arises for example converting a *Src* to a *CoSrc*, as we have seen in the example above, when writing the file co-source.

In general, given a *Schedule*, we can implement the following two conversions:

$srcToCoSrc :: Schedule\ a \to Src\ a \multimap CoSrc\ a$
$coSnkToSnk :: Schedule\ a \to CoSnk\ a \multimap Snk\ a$

We define a *Schedule* as the reconciliation between a source and a co-sink:

type $Schedule\ a = Source\ a \multimap CoSnk\ a \multimap Eff$

Implementing the conversions is then straightforward:

$srcToCoSrc\ strat\ s0\ s = s0\ \$\ Cont\ (flip\ strat\ s)$
$coSnkToSnk\ strat\ s0\ s = s\ \$\ Cont\ (flip\ strat\ s0)$

What are possible scheduling strategies? The simplest, and most natural one is sequential execution. Namely, looping through both sources and match the consumptions/productions element-wise, as follows.

```
sequentially :: Drop a ⇒ Schedule a
sequentially Nil s = s Full
sequentially (Cons x xs) s = s $ Cont $ λs′ → case s′ of
    Cons x′ xs′ → do
       x′ x
       xs $ Cont $ λt → sequentially t xs′
    Nil → drop x (xs Full)
```

For most streams, sequential execution is the only sensible schedule: indeed, sources and sinks often expect their effects to be run in the order prescribed by the stream. Swapping the arguments to $<>$ in the above means that *Full* effects will be run first, spelling disaster.

However, in certain cases running effects out of order may make sense. If the monoid of effects is commutative (or if the programmer is confident that execution order does not matter) one can shuffle the order of execution of effects. This re-ordering can be taken advantage of to run effects concurrently, as follows:

```
concurrently :: Drop a ⇒ Schedule a
concurrently Nil s = s Full
concurrently (Cons x xs) s = s $ Cont $ λs′ → case s′ of
    Cons x′ xs′ → do
       forkIO (x′ x)
       xs $ Cont $ λt → concurrently t xs′
    Nil → drop x (xs Full)
```

The above strategy is useful if the production or consumption of elements is expensive and distributable over computation units. While the above implementation naively spawns a thread for every element, in reality one should most likely divide the stream into chunks before spawning threads. Because strategies are separate components, a bad choice is easily remedied to by swapping one strategy for another.

Buffering Consider now the dual situation where we have two streams and both of them want to control the execution flow, for example when converting a *CoSrc* to a *Src*. The conciliation can only be implemented by running both streams in concurrent threads, and have them communicate via some form of buffer. Using the primitives seen so far, one can implement a file-based buffer, as follows:

```
fileBuffer :: String → CoSrc String ⊸ Src String
fileBuffer tmpFile f g = do
    h′ ← openFile tmpFile WriteMode
    forkIO $ f (coFileSink h′)
    h ← openFile tmpFile ReadMode
    hFileSrc h g
```

It should be stressed that buffering is a risky operation: it is a balancing act between the amount of resources one is prepared to allocate and the dynamic behaviours of the streams. In fact, our simple file-buffer above is likely to fail if the intermediate file is a regular file. Indeed, the reader may for example be faster than the writer and reach an end of file prematurely. One can use instead a UNIX pipe, but then one then faces the issue that pipes are of fixed maximum size: if the writer outputs data in too large chunks, a deadlock will occur.

If one is prepared to use unbounded memory, one may use Concurrent Haskell channels as a buffering means:

```
chanCoSnk :: Chan a → CoSnk a
chanCoSnk _ Full = return ()
chanCoSnk h (Cont c) = c (Cons (writeChan h)
                                (chanCoSnk h))

chanSrc :: Chan a → Src a
chanSrc _ Full = return ()
chanSrc h (Cont c) = do x ← readChan h
                        c (Cons x $ chanSrc h)

chanBuffer :: CoSrc a ⊸ Src a
chanBuffer f g = do
    c ← newChan
    forkIO $ f (chanCoSnk c)
    chanSrc c g
```

Note that it is easy to create a bounded channel-based buffer, by guarding the writes with a semaphore.

```
chanCoSnk′ :: Chan a → QSem → CoSnk a
chanCoSnk′ _ _ Full = return ()
chanCoSnk′ h s (Cont c) = c (Cons write
                                (chanCoSnk′ h s))
where write x = do waitQSem s
                   writeChan h x

chanSrc′ :: Chan a → QSem → Src a
chanSrc′ _ _ Full = return ()
chanSrc′ h s (Cont c) = do x ← readChan h
                           signalQSem s
                           c (Cons x $ chanSrc′ h s)

boundedChanBuffer :: Int → CoSrc a ⊸ Src a
boundedChanBuffer n f g = do
    c ← newChan
    s ← newQSem n
    forkIO $ f (chanCoSnk′ c s)
    chanSrc′ c s g
```

In certain situations (for example for a stream yielding a status whose history does not matter, like mouse positions) one may want to ignore all but the latest datum. In this case a single memory cell can serve as buffer:

```
varCoSnk :: IORef a → CoSnk a
varCoSnk _ Full       = return ()
varCoSnk h (Cont c) = c (Cons (writeIORef h)
                                (varCoSnk h))

varSrc :: IORef a → Src a
varSrc _ Full = return ()
varSrc h (Cont c) = do x ← readIORef h
                       c (Cons x $ varSrc h)

varBuffer :: a → CoSrc a ⊸ Src a
varBuffer initialValue f g = do
    c ← newIORef initialValue
```

$forkIO \ \$ \ f \ (varCoSnk \ c)$
$varSrc \ c \ g$

The above buffering operations work on sources, but they can be generically inverted to work on sinks, as follows.

$flipBuffer :: (\forall a.CoSrc \ a \multimap Src \ a) \to Snk \ b \multimap CoSnk \ b$
$flipBuffer \ f \ s = f \ (s \circ nnElim)$

6.4 Summary

In sum, we can classify streams according to polarity:

- Pull: source and co-sinks
- Push: sinks and co-sources

We then have three situations when composing stream processors:

1. Matching polarities (one pull, one push). In this case behavior is synchronous; no concurrency appears.
2. Two pull streams. In this case an explicit loop must process the streams. If effects commute, the programmer may run effects out of order, potentially concurrently.
3. Two push streams. In this case the streams must run in independent threads, and the programmer needs to make a choice for the communication buffer. One needs to be careful: if the buffer is too small a deadlock may occur.

Therefore, when programming with streams, one should consume push types when one can, and pull ones when one must. Conversely, one should produce pull types when one can, and push ones when one must.

6.5 App: idealized echo server

We finish the exposition of asynchronous behavior with a small program sketching the skeleton of a client-server application. This is a small server with two clients, which echoes the requests of each client to both of them.

The server communicates with each client via two streams, one for inbound messages, one for outbound ones. We want each client to be able to send and receive messages in the order that they like. That is, from their point of view, they are in control of the message processing order. Hence a client should have a co-sink for sending messages to the server, and a source for receiving them. On the server side, types are dualized and thus, a client is represented by a pair of a co-source and a sink:

type $Client \ a = (CoSrc \ a, Snk \ a)$

For simplicity we implement a chat server handling exactly two clients.

The first problem is to multiplex the inputs of the clients. In the server, we do not actually want any client to be controlling the processing order. Hence we have to multiplex the messages in real time, using a channel (note the similarity with $chanBuffer$):

$bufferedDmux :: CoSrc \ a \multimap CoSrc \ a \multimap Src \ a$
$bufferedDmux \ s1 \ s2 \ t = \textbf{do}$
$\quad c \leftarrow newChan$
$\quad forkIO \ \$ \ s1 \ (chanCoSnk \ c)$

$\quad forkIO \ \$ \ s2 \ (chanCoSnk \ c)$
$\quad chanSrc \ c \ t$

We then have to send each message to both clients. This may be done using the following effect-free function, which forwards everything sent to a sink to its two argument sinks.

$collapseSnk' :: (a \multimap (b, c)) \to Snk' \ b \multimap Snk' \ c \multimap Snk' \ a$
$collapseSnk' \ \_ \quad t1 \ t2 \ Nil = t1 \ Nil \Diamond t2 \ Nil$
$collapseSnk' \ dup \ t1 \ t2 \ (Cons \ x \ xs)$
$\quad = t1 \ (Cons \ y \ \$ \ \lambda c1 \to$
$\quad\quad t2 \ (Cons \ z \ \$ \ tee' \ dup \ xs \ c1))$
$\quad \textbf{where} \ (y, z) = dup \ x$

$tee' :: (a \multimap (b, c)) \to Src \ a \to Sink \ b \to Src \ c$
$tee' \ deal \ s1 \ t1 \ Full = s1 \ Full \Diamond empty \ t1$
$tee' \ deal \ s1 \ Full \ t2 = s1 \ Full \Diamond empty \ t2$
$tee' \ deal \ s1 \ (Cont \ t1) \ (Cont \ t2) = s1 \ \$ \ Cont \ \$ \ collapseSnk' \ deal \ t1 \ t2$

The server can then be defined by composing the above two functions.

$server :: (a \multimap (a, a)) \multimap Client \ a \multimap Client \ a \multimap Eff$
$server \ dup \ (i1, o1) \ (i2, o2) = fwd \ (bufferedDmux \ i1 \ i2)$
$\quad\quad\quad\quad\quad\quad\quad\quad (collapseSnk \ dup \ o1 \ o2)$

7 Related Work

7.1 Polarities, data structures and control

One of keys ideas formalized in this paper is to classify streams by polarity. The push polarity (sinks, co-sources) controls the execution thread, whereas the pull one (sources, co-sinks) provide data. This idea has been leveraged to bring efficient array programming facilities to functional programming [1, 3, 7].

This concept is central in the literature on Girard's linear logic [21, 29]. However, in the case of streams, this idea dates back at least to Jackson [13] (Kay [14] gives a good summary of Jacksons' insight).

Our contribution is to bring this idea to stream programming in Hask-LL. (While duality was used for Haskell array programming, it has not been taken advantage for stream programming, and there was no emphasis on linearity.) We believe that our implementation brings together the practical applications that Jackson intended, while being faithful to the theoretical foundations in logic, via the double-negation embedding.

7.2 Transducers

Shivers and Might [24] introduces a transducer library which enables fusing the transducers to avoid the overhead of composition. Transducers are defined on top of channels, a recursive datatype reminiscent of our $Source$ and $Sink$. In particular they use a type for continuations similar to N. However, a channel is just one type and does not exhibit the duality that our sources and sinks provide. Their library do feature a notion of sources and sinks but they are not the main abstraction. They are simply aids for providing and consuming data from transducers, respectively. Transducers are "affine"; they can be used at most once. The reason

linearity is not required is that none of the transducers are effectful, the library only provides pure transducers. Therefore it is also not concerned with releasing resources in a timely fashion.

7.3 Iteratees

We consider that the state of the art in Haskell stream processing is embodied by Kiselyov's iteratees [2012].

The type for iteratees can be given the following definitions:

data $I\ s\ m\ a = Done\ a\ |\ GetC\ (Maybe\ s \to m\ (I\ s\ m\ a))$

An iteratee $I\ s\ m\ a$ roughly corresponds to a sink of s which also returns an a — but it uses a monad m rather than a monoid Eff for effects.

The above type contains a continuation in the $GetC$ constructor. Therefore, one must be careful about discarding or duplicating iteratees. Hence, such libraries typically provide higher-level interfaces to discourage non-linear usages.

A first advantage of our approach is the formulation and emphasis on the linearity constraint, which is central to correct use of effectful continuations. It appears that variants of iteratees (including the *pipes* library) make the representation abstract, but at the cost of a complex interface for programming them. By stating the linearity requirement no complex abstract API is necessary to guarantee safety.

A second advantage of our library is that effects are not required to be monads. Indeed, the use of continuations already provide the necessary structure to combine computations (recall in particular that double negation is already a monad). We believe that having a single way to bind intermediate results (continuations vs. both continuations and monads) is a simplification in design, which may make our library more approachable.

The presence of source and sinks also clarifies how to build complex types programs from basic blocks. Indeed, iteratee-based libraries make heavy use of the following types:

type $Enumerator\ el\ m\ a = I\ el\ m\ a \to m\ (I\ el\ m\ a)$
type $Enumeratee\ elo\ eli\ m\ a =$
    $I\ eli\ m\ a \to I\ elo\ m\ (I\ eli\ m\ a)$

It is our understanding that these types make up for the lack of explicit sources by putting iteratees (sinks) on the left-hand-side of an arrow. Enumerators are advantageously replaced by sources, and enumeratees by simple functions from source to source (or sink to sink).

A third advantage of our approach is that the need for buffering (or the scheduling opportunities) are clearly indicated by the type system, as mismatching polarities.

In more recent work Kiselyov et al. [19] present a continuation-based pretty printer, which fosters a more stylized used of continuations, closer to what we advocate here. Producers and consumers (sources and sinks) are defined more simply, using types which correspond more directly to negations:

type $GenT\ e\ m = ReaderT\ (e \to m\ ())\ m$
type $Producer\ m\ e = GenT\ e\ m\ ()$
type $Consumer\ m\ e = e \to m\ ()$
type $Transducer\ m1\ m2\ e1\ e2 =$
    $Producer\ m1\ e1 \to Producer\ m2\ e2$

Yet, in that work, linearity is only briefly mentioned; the use of a monad rather than monoid persists; and mismatching polarities are not discussed, let alone taken advantage of.

Several production-strength libraries have been built upon the concept of iteratees, including *pipes* [11], *conduits* [25] and *machines* [20]. While we focus our comparison with iteratees, most of our analysis carries to the production libraries. There is additionally a large body of non peer-reviewed literature discussing and analyzing either iteratees or its variants. The proliferation of libraries for IO streaming in Haskell indicates that a unifying foundation for them is needed, and we hope that the present paper provides a basis for such a foundation.

7.4 Feldspar monadic streams

Feldspar, a DSL for digital signal processing, has a notion of streams built on monads [2, 27]. In Haskell the stream type can be written as follows:

type $Stream\ a = IO\ (IO\ a)$

Intuitively the outer monad can be understood as performing initialization which creates the inner monadic computation. The inner computation is called iteratively to produce the elements of the stream.

Compared to the representation in the present paper, the monadic streams only has one form of stream, corresponding to a source. Also, there is no support for timely release of resources, such things need to be dealt with outside of the stream framework. Additionally, even conceptually effect-free streams rely on running IO effects.

7.5 Session Types

In essence our pair of types for stream is an encoding of a protocol for data transmission. This protocol is readily expressible using linear types, following the ideas of Wadler [28] and Caires et al. [5]:

$Source\ a = 1 \oplus (a \otimes N\ (Sink\ a))$
$Sink\ a = 1 \oplus N\ (Source\ a)$

For the translation to Haskell, we have chosen to use a lightweight encoding, assuming linearity of effectful variables; arguing at the same time for support of linearity in future Haskell versions. Yet, other encodings could be chosen. For example, we could have used the technique of Pucella and Tov (Haskell session types with almost no class), which does not require abiding to linearity.

8 Future Work

As we see it, a natural next step for the present work is to show that intermediate sources and sinks can be deforested. As it stands, we believe that a standard approach [8, 9, 26] should work: 1. encode sources (and sinks) as non-recursive data types 2. show that standard evaluation rules remove the intermediate occurrences of the encoded types. However, this work has not been carried out yet.

The duality principle exposed here as already been taken advantage of to support fusible array types [1, 3]. The present paper has shown how to support effectful stream computations. One would naturally think that the same principle can be applied to other lazily-evaluated data structures, such as

the game trees discussed by Hughes [12]: as far as we know this remains to be investigated.

9   Conclusion

We have cast a new light on the current state of coroutine-based computation in Haskell, which we have done by drawing inspiration from classical linear logic. We have further shown that the concepts of duality and polarity provide design principles to structure continuation-based code. In particular, we have shown that mismatches in polarity correspond to buffers and control structures, depending on the kind of mismatch.

Using effectful continuations is not a new idea; in fact it was the standard way of writing effectful programs in Haskell 1.2. Later versions of Haskell switched to a monadic approach. However, given the issues outlined in the introduction, and especially the error-prone character of lazy monadic IO, many libraries have reverted to explicitly using co-routines.

A possible reason for selecting monads over co-routines is that monads are rooted in solid theory (categories). However, we hope to have shown that co-routines are also rooted in solid theory, namely linear logic. If Haskell had support for linear types, co-routines could be used safely, without the quirks of lazy IO.

Acknowledgments

References

[1] J. Ankner and J. D. Svenningsson. An edsl approach to high performance haskell programming. In *Proc. of the 2013 ACM SIGPLAN symposium on Haskell*, pages 1–12. ACM, 2013.

[2] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 169–178. IEEE, 2010.

[3] J.-P. Bernardy, V. López Juan, and J. Svenningsson. Composable efficient array computations using linear types, 2015. Submitted to ICFP 2015. Draft available online.

[4] J.-P. Bernardy, A. Spiwack, M. Boespflug, R. Newton, and S. Peyton Jones. Retrofitting linear types, 2017. Draft available online.

[5] L. Caires, F. Pfenning, and B. Toninho. Towards concurrent type theory. In *Proc. of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 1–12. ACM, 2012.

[6] K. Claessen. Parallel parsing processes. *J. of Funct. Prog.*, 14 (6):741–757, 2004.

[7] K. Claessen, M. Sheeran, and B. J. Svensson. Expressive array constructs in an embedded gpu kernel programming language. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, pages 21–30. ACM, 2012.

[8] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Haskell*, pages 315–326. ACM, 2007.

[9] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Proc. of FPCA*, pages 223–232. ACM, 1993.

[10] J.-Y. Girard. Linear logic. *Theor. Comp. Sci.*, 50(1):1–101, 1987.

[11] G. Gonzalez. The pipes package, 2015.

[12] J. Hughes. Why funct. prog. matters. *Comp. J.*, 32(2):98–107, 1989.

[13] M. A. Jackson. *Principles of Program Design*. Academic Press, Inc., 1975.

[14] M. Kay. You pull, I'll push: on the polarity of pipelines. In *Proceeding of Balisage: The Markup Conf.*, 2008.

[15] O. Kiselyov. Combining monadic regions and iteratees. http://okmij.org/ftp/Streams.html#regions, Jan 2012.

[16] O. Kiselyov. Iteratees. In *Funct. and Logic Prog. - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proc.*, pages 166–181, 2012.

[17] O. Kiselyov. Lazy io breaks equational reasoning, 2013. Manuscript available on the author's web page.

[18] O. Kiselyov and C.-c. Shan. Lightweight monadic regions. In *Haskell Symposium.* ACM, 2008.

[19] O. Kiselyov, S. L. Peyton Jones, and A. Sabry. Lazy v. yield: Incremental, linear pretty-printing. In *Prog. Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proc.*, pages 190–206, 2012.

[20] E. A. Kmett, R. Bjarnason, and J. Cough. The machines package, 2015.

[21] O. Laurent. *Etude de la polarisation en logique*. Thèse de doctorat, Université Aix-Marseille II, 2002.

[22] P.-A. Melliès and N. Tabareau. Resource modalities in tensor logic. *Ann. Pure Appl. Logic*, 161(5):632–653, 2010.

[23] G. Munch-Maccagnoni. Formulae-as-types for an involutive negation. In *Joint Meeting of the Twenty-Third EACSL Annual Conf. on Comp. Sci. Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Comp. Sci. (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 70:1–70:10, 2014.

[24] O. Shivers and M. Might. Continuations and transducer composition. *ACM SIGPLAN Notices*, 41(6):295–307, 2006.

[25] M. Snoyman. The conduit package, 2015.

[26] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proc. of ICFP 2002*, pages 124–132. ACM, 2002.

[27] J. Svenningsson, E. Axelsson, A. Persson, and P. A. Jonsson. Efficient monadic streams. In *Presented at Trends in Functional Programming*, 2015.

[28] P. Wadler. Propositions as sessions. In *Proc. of ICFP 2012*, ICFP '12, pages 273–286. ACM, 2012.

[29] N. Zeilberger. *The logical basis of evaluation order and pattern-matching*. PhD thesis, Carnegie Mellon University, 2009.