On the Duality of Streams

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

Jean-Philippe Bernardy Josef Svenningsson
Chalmers University of Technology and University of Gothenburg
bernardy,josefs at chalmers.se

Abstract

We present a novel stream-programming library for Haskell. As other coroutine-based stream libraries, our library allows synchronous execution, which implies that effects are run in lockstep and no buffering occurs.

A novelty of our implementation is that it allows to locally introduce buffering or re-scheduling of effects. The buffering requirements (or re-scheduling opportunities) are indicated by the type-system.

Our library is based on a number of design principles, adapted from the theory of Girard's Linear Logic. These principles are applicable to the design of any Haskell structure where resource management (memory, IO, ...) is critical.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.3.3 [*Language Constructs and Features*]: Coroutines

Keywords Streams, Continuations, Linear Types

1. Introduction

As **?** 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 list transformation functions. Indeed, while 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 **?** has shown). For example, one may rightfully expect[1] that both following programs have the same behavior:

$$\begin{aligned} main = \mathbf{do}\ & inFile \leftarrow openFile\ \texttt{"foo"}\ ReadMode \\ & contents \leftarrow hGetContents\ inFile \\ & putStr\ contents \\ & hClose\ inFile \end{aligned}$$

$$\begin{aligned} main = \mathbf{do}\ & inFile \leftarrow openFile\ \texttt{"foo"}\ ReadMode \\ & contents \leftarrow hGetContents\ inFile \\ & hClose\ inFile \\ & putStr\ contents \end{aligned}$$

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 force 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. 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 mimicking the computational behavior of Girard's linear logic (**?**) in Haskell. In fact, one way to read this paper is as an advocacy for linear types support in Haskell. While Kiselyov's *iteratees* (**?**) already solves the issues described above, our grounding in linear logic yields a rich structure for types for data streams, capturing various production and consumption patterns.

First, the 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, and give the guarantee that the composition does not allocate more memory than the sum of its components.

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

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

Second, the type driving the consumption of elements is called a sink (Snk). For example, the standard output is naturally given a sink type:

$stdoutSnk :: Snk\ String$

Using it, we can implement the printing of a file as follows, and guarantee the timely release of resources, even in the presence of exceptions:

$main = fileSrc$ `"foo"` $`fwd` stdoutSnk$

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:

$fileSrc :: FilePath \to Src\ String$
$fwd :: Src\ a \to Snk\ a \to IO\ ()$

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. We support in particular data sources with push-flavor, called co-sources ($CoSrc$). Co-sources are useful for example when a data stream needs precise control over the execution of effects it embeds (sec 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:

$f :: Src\ a \to CoSrc\ b$
$g :: Src\ b \to Src\ c$
$h = g \circ buffer \circ f$

The contributions of this paper are

- The formulation of principles for compositional resource-aware programming in Haskell (resources include memory and files). The principles are linearity, duality, and polarization. While 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 Haskell 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:

$\mathbf{type}\ N\ a = a \to Eff$

A shortcut for double negations is also convenient.

$\mathbf{type}\ NN\ a = N\ (N\ a)$

The basic idea (imported from classical logic) pervading this paper is that 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 $N\alpha$. In this paper 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[2]. We can come close enough though. First, double negations can always be introduced, using the $shift$ operator:

$shift :: a \to NN\ a$
$shift\ x\ k = k\ x$

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:

$unshift :: N\ (NN\ a) \to N\ a$
$unshift\ k\ x = k\ (shift\ x)$

The above two functions are the $return$ and $join$ of the double negation monad[3]; 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. Indeed, single negations play a central role in our approach, and 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 is also the natural choice when interpreting the original linear logic of **?**.

The pure logic makes no requirement on effects, 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 of effects. (This structure is standard to interpret the HALT and MIX rules in linear logic (**??**))

For users of the stream library, *Eff* will remain an abstract monoid. However in this paper we will 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 Haskell, we must pick $Eff = IO()$, and ensure that *Eff* can be treated as a monoid.

$\mathbf{type}\ Eff = IO\ ()$

$\mathbf{instance}\ Monoid\ Eff\ \mathbf{where}$
$mempty = return\ ()$
$(\diamond) = (\gg)$

[2] Even though **?** achieves an involutive negation in an intuitionistic language, he does so by stack manipulation, which is not available in Haskell.

[3] for $join$, substitute $N\,a$ for a

The parts of the code which know about $Eff = IO()$ must be carefully written. The type system provides no particular guarantees about such code. These IO-interacting functions do not interpret any standard fragment of linear logic: they are non-standard extensions of its model.

3. Streams

Our guiding design principle is duality. This 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 will be given the following type:

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

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 have been written as follows:

$$simple :: Src\ a \to Src\ a$$

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

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

Familiarity with duality will be crucial in the later sections of this paper.

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.

$$\begin{array}{l}\mathbf{data}\ Source\ a = Nil \mid Cons\ a\ (N\ (Sink\ \ a)) \\ \mathbf{data}\ Sink\ \ a = Full \mid Cont\ \ (N\ (Source\ a))\end{array}$$

Producing a sink means to select if 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 encounters an exception.

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*.

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 will typically be 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). In the rest of the section we precisely define the condition that programs need to respect in order to safely use our streams.

The first notion that we need to define is that of an effectful type:

- The type Eff is effectful
- A function type is effectful if the co-domain is effectful
- A product type is effectful if any of its operands is effectful
- A sum type is effectful if any of its operands is effectful
- A type variable is not effectful

Further, we say that a variable with an effectful type is itself effectful.

The linearity convention is then respected iff:

1. No effectful variable may be duplicated or shared. In particular, if passed as an argument to a function it may not be used again.
2. Every effectful variable must be consumed (or passed to a function, which will be in charged of consuming it).
3. A type variable α can not be instantiated to an effectful type.

In this paper, the linearity convention is enforced by manual inspection. Manual inspection is unreliable, but weather the linearity convention is respected can be algorithmically decided. (See sec. 8)

The third restriction (instantiation of type-variables) means that effectful types cannot be used in standard polymorphic Haskell functions. This is a severe restriction, but it gives enough leeway to implement a full-fledged stream library, as we do below. (Yet some approached to lift this limitation have been proposed, e.g. by **?**.)

One might think that the above restriction fails to take into account captured environments in functions. Indeed, one can write the following function, which may be duplicated, but runs linear effects.

$$\begin{array}{l}oops :: (() \to Eff) \to IO\ Bool \\ oops\ k = \mathbf{do}\ ignore \leftarrow k\ () \\ \quad return\ True\end{array}$$

However, writing such a function requires to know that $Eff = IO()$, and is therefore disallowed by the Haskell type system in user code, where Eff is kept abstract. (The $mappend$ function may combine two effects in one, but not discard or duplicate them.)

3.2 Basics

We begin by presenting three basic function to manipulate $Source$ and $Sink$: one to read from sources, one to write to sinks, and one to connect sources and sinks.

Reading One may want to provide 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.

$$\begin{array}{l}await :: Source\ a \to Eff \to (a \to Source\ a \to Eff) \to Eff \\ await\ Nil\ eof\ \_ = eof \\ await\ (Cons\ x\ cs)\ \_\ k = cs\ \$\ Cont\ \$\ \lambda xs \to k\ x\ xs\end{array}$$

However, the above function breaks the linearity invariant, so we will refrain to use it as such. The pattern that it defines is still useful: it is valid when the second and third argument consume the same set of variables. Indeed, this condition is often satisfied.

Writing One can send data to a sink. If the sink is full, the data is ignored. The third argument is a continuation getting the "new" sink, that obtained after the "old" sink has consumed the data.

$$\begin{array}{l}yield :: a \to Sink\ a \to (Sink\ a \to Eff) \to Eff \\ yield\ x\ (Cont\ c)\ k = c\ (Cons\ x\ k) \\ yield\ \_\ Full\ k = k\ Full\end{array}$$

Forwarding One can forward the data from a source to a sink, as follows. The effect generated by this operation is the combined effect of all productions and consumptions on the stream.

$$\begin{array}{l}forward :: Source\ a \to Sink\ a \to Eff \\ forward\ s\ (Cont\ s') = s'\ s \\ forward\ Nil\ Full = mempty \\ forward\ (Cons\ \_\ xs)\ Full = xs\ Full\end{array}$$

3.3 Baking in negations: exercise in duality

Programming with $Source$ and $Sink$ explicitly is inherently continuation-heavy: negations must be explicitly added in many places. This

style is somewhat inconvenient; therefore, we will use instead pre-negated versions of sources and sink:

```
type Src  a = N (Sink a)
type Snk a = N (Source a)
```

These definitions have the added advantage to perfect the duality between sources and sinks, while not restricting the programs one can write. Indeed, one can access the underlying structure as follows:

```
onSource :: (Src  a → t) → Source a → t
onSink   :: (Snk a → t) → Sink   a → t

onSource f s = f (λt → forward s t)
onSink   f t = f (λs → forward s t)
```

And, while a negated *Sink* cannot be converted to a *Source*, all the following conversions are implementable:

```
unshiftSnk :: N (Src a) → Snk a
unshiftSrc :: N (Snk a) → Src a
shiftSnk :: Snk a → N (Src a)
shiftSrc :: Src a → N (Snk a)

unshiftSnk = onSource
unshiftSrc = onSink
shiftSnk k kk = kk (Cont k)
shiftSrc k kk = k (Cont kk)
```

A different reading of the type of *shiftSrc* reveals that it implements forwarding of data from *Src* to *Snk*:

```
fwd :: Src a → Snk a → Eff
fwd = shiftSrc
```

In particular, one can flip sink transformers to obtain source transformers, and vice versa.

```
flipSnk :: (Snk a → Snk b) → Src b → Src a
flipSnk f s = shiftSrc s ∘ onSink f

flipSrc :: (Src a → Src b) → Snk b → Snk a
flipSrc f t = shiftSnk t ∘ onSource f
```

Flipping allows to choose the most convenient direction to implement, and get the other one for free. Consider as an example the implementation of the mapping functions:

```
mapSrc :: (a → b) → Src  a → Src  b
mapSnk :: (b → a) → Snk a → Snk b
```

Mapping sources is defined by flipping mapping of sinks:

```
mapSrc f = flipSnk (mapSnk f)
```

Sink mapping 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)
  = snk (Cons (f a) (mapSrc f s))
```

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

```
nnIntro :: Src a → Src (NN a)
nnIntro = mapSrc shift

nnElim' :: Snk (NN a) → Snk a
nnElim' = mapSnk shift
```

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

```
nnElim :: Src (NN a) → Src a
nnIntro' :: Snk a → Snk (NN a)

nnElim = flipSnk nnIntro'
nnIntro' k Nil = k Nil
nnIntro' k (Cons x xs) = x $ λx' → 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. In fact, 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 sink' = forward Nil sink'

cons :: a → Src a → Src a
cons a s s' = yield a s' s

tail :: Src a → Src a
tail = flipSnk $ λt s → case s of
  Nil → t Nil
  Cons _ xs → fwd xs t
```

(Taking just the head is not meaningful due to the linearity constraint)

Dually, the full sink is simply

```
plug :: Snk a
plug source' = forward source' Full
```

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.

```
takeSrc :: Int → Src  a → Src  a
takeSnk :: Int → Snk a → 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 i = flipSnk (takeSnk i)

takeSnk _ s Nil = s Nil
takeSnk 0 s (Cons _ s') = s Nil <> s' Full
takeSnk i s (Cons a s') = s (Cons a (takeSrc (i − 1) s'))
```

4.2 Algebraic structure

Source and sinks form a monoid under concatenation:

```
instance Monoid (Src a) where
  (<>) = appendSrc
  mempty = empty

instance Monoid (Snk a) where
  (<>) = appendSnk
  mempty = plug
```

We have already encountered the units ($empty$ and $plug$); 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 $forwardThenSnk$.

$$
\begin{aligned}
&appendSrc :: Src\ a \to Src\ a \to Src\ a\\
&appendSrc\ s1\ s2\ Full = s1\ Full <\!\!> s2\ Full\\
&appendSrc\ s1\ s2\ (Cont\ s)\\
&\quad = s1\ (Cont\ (forwardThenSnk\ s\ s2))\\
\\
&forwardThenSnk :: Snk\ a \to Src\ a \to Snk\ a\\
&forwardThenSnk\ snk\ src\ Nil = fwd\ src\ snk\\
&forwardThenSnk\ snk\ src\ (Cons\ a\ s)\\
&\quad = snk\ (Cons\ a\ (appendSrc\ s\ src))
\end{aligned}
$$

Sinks can be appended is a similar fashion.

$$
\begin{aligned}
&appendSnk :: Snk\ a \to Snk\ a \to Snk\ a\\
&appendSnk\ s1\ s2\ Nil = s1\ Nil <\!\!> s2\ Nil\\
&appendSnk\ s1\ s2\ (Cons\ a\ s)\\
&\quad = s1\ (Cons\ a\ (forwardThenSrc\ s2\ s))\\
\\
&forwardThenSrc :: Snk\ a \to Src\ a \to Src\ a\\
&forwardThenSrc\ s2 = flipSnk\ (appendSnk\ s2)
\end{aligned}
$$

The operations $forwardThenSnk$ and $forwardThenSrc$ are akin to making the difference of sources and sinks, thus we find it convenient to give them the following aliases:

$$
\begin{aligned}
&(-?) :: Snk\ a \to Src\ a \to Snk\ a\\
&t\ -?\ s = forwardThenSnk\ t\ s\\
\\
&(-!) :: Snk\ a \to Src\ a \to Src\ a\\
&t\ -!\ s = forwardThenSrc\ t\ s\\
\\
&\mathbf{infixr}\ -!\\
&\mathbf{infixl}\ -?
\end{aligned}
$$

Appending and differences interact in the expected way: the following observational equalities hold:

$$
\begin{aligned}
&t\ -?\ (s1 <\!\!> s2) \equiv t\ -?\ s2\ -?\ s1\\
&(t1 <\!\!> t2)\ -!\ s \equiv t1\ -!\ t2\ -!\ s
\end{aligned}
$$

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.)

Monad It is possible to write a monad instance for sources. However, it violates the linearity convention. Consider the type of $join$:

$$join :: Src\ (Src\ a) \to Src\ a$$

The type parameter of the outer source has been instantiated with an effectful type. Allowing monads would require a more complex type system than the linearity convention we employ in this paper. We have yet to find a need for a monad instance when programming with our stream library.

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 in the appendix.

Zip two sources, and the dual.

$$
\begin{aligned}
&zipSrc :: Src\ a \to Src\ b \to Src\ (a, b)\\
&forkSnk :: Snk\ (a, b) \to Src\ a \to Snk\ b
\end{aligned}
$$

Zip two sinks, and the dual.

$$
\begin{aligned}
&forkSrc :: Src\ (a, b) \to Snk\ a \to Src\ b\\
&zipSnk :: Snk\ a \to Snk\ b \to Snk\ (a, b)
\end{aligned}
$$

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

$$
\begin{aligned}
&scanSrc :: (b \to a \to b) \to b \to Src\ a \to Src\ b\\
&scanSnk :: (b \to a \to b) \to b \to Snk\ b \to Snk\ a
\end{aligned}
$$

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

$$
\begin{aligned}
&foldSrc' :: (b \to a \to b) \to b \to Src\ a \to NN\ b\\
&foldSnk' :: (b \to a \to b) \to b \to N\ b \to Snk\ a
\end{aligned}
$$

Drop some elements from a source, and the dual.

$$
\begin{aligned}
&dropSrc :: Int \to Src\ a \to Src\ a\\
&dropSnk :: Int \to Snk\ a \to Snk\ a
\end{aligned}
$$

Convert a list to a source, and vice versa.

$$
\begin{aligned}
&fromList :: [a] \to Src\ a\\
&toList :: Src\ a \to NN\ [a]
\end{aligned}
$$

Split a source in lines, and the dual.

$$
\begin{aligned}
&linesSrc :: Src\ Char \to Src\ String\\
&unlinesSnk :: Snk\ String \to Snk\ Char
\end{aligned}
$$

Consume elements until the predicate is reached; then the sink is closed.

$$untilSnk :: (a \to Bool) \to Snk\ a$$

Interleave two sources, and the dual.

$$
\begin{aligned}
&interleave :: Src\ a \to Src\ a \to Src\ a\\
&interleaveSnk :: Snk\ a \to Src\ a \to Snk\ a
\end{aligned}
$$

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

$$tee :: Src\ a \to Snk\ a \to Src\ a$$

Filter a source, and the dual.

$$
\begin{aligned}
&filterSrc :: (a \to Bool) \to Src\ a \to Src\ a\\
&filterSnk :: (a \to Bool) \to Snk\ a \to Snk\ a
\end{aligned}
$$

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

$$
\begin{aligned}
&unchunk :: Src\ [a] \to Src\ a\\
&chunkSnk :: Snk\ a \to Snk\ [a]
\end{aligned}
$$

4.4 App: Stream-Based Parsing

To finish with effect-free function, we give an example of a complex stream processor, which turns source of unstructured data into a source of structured data, given a parser. 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 **?**. 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 $Result$.

$$
\begin{aligned}
\mathbf{data}\ &P\ s\ res = Sym\ (Maybe\ s \to P\ s\ res)\\
&\quad \mid Fail\\
&\quad \mid Result\ res
\end{aligned}
$$

A parser is producing the double negation of a:

$$\mathbf{newtype}\ Parser\ s\ a = P\ (\forall res.(a \to P\ s\ res) \to P\ s\ res)$$

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) y = Result res
weave x (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 q@(P p0) = flipSnk $ scan $ p0 $ λx → Result x
where
  scan :: P s a → Snk a → Snk s
  scan (Result res) ret    xs = ret
    (Cons res $ parse q $ forward xs)
  scan Fail         ret    xs = ret Nil <> forward xs Full
  scan (Sym f)      mres xs = case xs of
    Nil        → scan (f Nothing) mres Nil
    Cons x cs → fwd cs (scan (f $ Just x) mres)
```

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
  s (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:

```
stdoutSnk :: 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 should 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 of sinks allows modular stream programming. In particular, it is easy to support proper finalization in the presence of exceptions, as the next section shows.

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. One would probably have a field in both the Nil and $Full$ constructors indicating the nature of the exception encountered, if any, but we leave it out in the proof of concept implementation presented in this paper.

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.

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 **?** can be used for this purpose, as outlined in **?**.

6. Synchronicity and Asynchronicity

One of the main benefits of streams as defined here is that the programming interface is (or appears to be) asynchronous, while the run-time behavior is synchronous. 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. Execution of connected sources and sinks is synchronous
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.

```
dmux' :: Src (Either a b) → Snk a → Snk b → Eff
```

We can implement this demultiplexing operation as follows:

```
dmux :: Source (Either a b) → Sink a → Sink b → Eff
dmux Nil ta tb = forward Nil ta <> forward Nil tb
dmux (Cons ab c) ta tb = case ab of
  Left a → c $ Cont $ λsrc' → case ta of
    Full → forward Nil tb <> plug src'
    Cont k → k (Cons a $ λta' → dmux src' ta' tb)
  Right b → c $ Cont $ λsrc' → case tb of
    Full → forward Nil ta <> plug src'
    Cont k → k (Cons b $ λtb' → dmux src' ta tb')

dmux' sab' ta' tb' =
  shiftSnk ta' $ λta →
  shiftSnk tb' $ λtb →
  shiftSrc sab' $ λsab →
  dmux sab 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 are 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, the choice falls to 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 case is a so-called additive conjunction. It is the dual of the $Either$ type: there is a choice, but this choice falls on the consumer rather than the producer of the data. Additive conjunction, written &, can be encoded by sandwiching $Either$ between two inversion of the control flow, thus switching the party who 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.)

We can then amend the type of multiplexing:

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

Unfortunately, we still cannot implement multiplexing typed as above. 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 → sb $ Cont $ λ(Cons b restb) → kb b)
  (error "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. Remembering that producing Na is equivalent to consuming a, thus 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 → CoSrc b → CoSrc (a & b)
mux' sa sb = unshiftSnk $ λtab → dmux' (nnElim tab) sa sb
```

We use the rest of the section to study the property of co-sources and co-sinks.

$CoSrc$ is a functor, and $CoSnk$ is a contravariant functor.

```
mapCoSrc :: (a → b) → CoSrc a → CoSrc b
mapCoSrc f = mapSnk (λb' → λa → b' (f a))

mapCoSnk :: (b → a) → CoSnk a → CoSnk b
mapCoSnk f = mapSrc (λb' → λa → b' (f a))
```

Elements of a co-source are access only "one at a time". That is, one cannot extract the contents of a co-source as a list. Attempting to implement this extraction looks as follows.

```
coToList :: CoSrc a → NN [a]
coToList k1 k2 = k1 $ Cons (λa → k2 [a]) (error "rest")
coToList k1 k2 = k2 $ (error "a?") : (error "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 and effectful co-sources and co-sinks. The first example shows how to provide a file as a co-source:

```
coFileSrc :: Handle → CoSrc String
coFileSrc h Nil = hClose h
coFileSrc h (Cons x xs) = do
  e ← hIsEOF h
  if e then do
      hClose h
      xs Full
    else do
      x' ← hGetLine h
      x x'                          -- (1)
      xs $ Cont $ coFileSrc h       -- (2)
```

Compared to *fileSrc*, the difference is that this function can decide the ordering of effects ran in a co-sink connected to it. That is, the lines (1) and (2) have no data dependency. Therefore they may be run in any order. (Blindly doing so is a bad idea though, as the *Full* action on the sink will be run before all other actions.) We will see in the next section how this situation generalizes.

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

```
coFileSink :: Handle → CoSnk String
coFileSink h Full = hClose h
coFileSink h (Cont c) = c (Cons (hPutStrLn h)
                                (coFileSink h))
```

Compared to *fileSnk*, the difference is that one 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, using co-sources and co-sinks shifts the flow of control from the sink to the 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 are interleaved.

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 forms of asynchronous programming within our framework. The main benefit of sticking to our framework in this case is that asynchronous behavior is cornered to the explicit usages of these primitives. That is, the benefits of synchronous programming still hold locally.

Scheduling When converting a *Src* to a *CoSrc* (or dually *CoSnk* to a *Snk*), we have two streams which are ready to respond to pulling of data from them. This means that effects must be scheduled explicitly, as we have seen an example above when manually converting the file source to a file co-source.

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

```
srcToCoSrc :: Schedule a → Src a → CoSrc a
coSnkToSnk :: Schedule a → CoSnk a → Snk a
```

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

```
type Schedule a = Source a → Source (N a) → Eff
```

Implementing the conversions is then straightforward:

```
srcToCoSrc strat s s0 = shiftSrc s $ λs1 → strat s1 s0
coSnkToSnk strat s s0 = shiftSrc s $ λs1 → strat s0 s1
```

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

```
sequentially :: Schedule a
sequentially Nil (Cons _ xs) = xs Full
sequentially (Cons _ xs) Nil = xs Full
sequentially (Cons x xs) (Cons x' xs') =
  x' x <> (shiftSrc xs $ λsa →
             shiftSrc xs' $ λsna →
             sequentially sa sna)
```

When effects are arbitrary IO actions, sequential execution is the only sensible schedule: indeed, the sources and sinks 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 :: Schedule a
concurrently Nil (Cons _ xs) = xs Full
concurrently (Cons _ xs) Nil = xs Full
concurrently (Cons x xs) (Cons x' xs') = do
  forkIO $ x' x
  (shiftSrc xs $ λsa →
    shiftSrc xs' $ λsna →
    concurrently sa sna)
```

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 will most likely want to 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 situation where one needs to convert from a *CoSrc* to a *Src* (or from a *Snk* to a *CoSnk*). Here, we have two streams, both of which want to control the execution flow. The conversion can only be implemented by running both streams in concurrent threads, and have them communicate via some form of buffer. A form of buffer that we have seen before is the file. Using it yields the following buffering implementation:

```
fileBuffer :: String → CoSrc String → Src String
fileBuffer tmpFile f g = do
```

```
  h' ← openFile tmpFile WriteMode
  forkIO $ fwd (coFileSink h') f
  h ← openFile tmpFile ReadMode
  hFileSrc h g
```

If the temporary file is a regular file, the above implementation is likely to fail. For example the reader may be faster than the writer and reach an end of file prematurely. Thus the temporary file should be a UNIX pipe. One then faces the issue that UNIX pipes are of fixed maximum size, and if the writer overshoots the capacity of the pipe, a deadlock will occur.

Thus, one may prefer to use Concurrent Haskell channels as a buffering means, as they are bounded only by the size of the memory and do not rely on any special feature of the operating system:

```
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 $ fwd (chanCoSnk c) f
  chanSrc c g
```

Note that it is easy to create a bounded buffer, by guarding the writes with a semaphore. In general there is no issue with blocking reads or writes. The implementation follows.

```
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 $ fwd (chanCoSnk' c s) f
  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 a f g = do
  c ← newIORef a
  forkIO $ fwd (varCoSnk c) f
  varSrc c g
```

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

```
flipBuffer :: (∀a. CoSrc a → Src a) → Snk b → CoSnk b
flipBuffer f s = f (nnIntro' s)
```

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 to 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 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 → CoSrc a → Src a
bufferedDmux s1 s2 t = do
  c ← newChan
```

```
forkIO $ fwd (chanCoSnk c) s1
forkIO $ fwd (chanCoSnk c) s2
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 :: Snk a → Snk a → Snk a
collapseSnk t1 t2 Nil = t1 Nil <> t2 Nil
collapseSnk t1 t2 (Cons x xs)
  = t1 (Cons x $ λc1 →
    t2 (Cons x $ λc2 →
      shiftSrc xs (collapseSnk (flip forward c1)
                               (flip forward c2))))
```

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

```
server :: Client a → Client a → Eff
server (i1, o1) (i2, o2) = fwd (bufferedDmux i1 i2)
                               (collapseSnk 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, CoSrc) controls the execution thread, whereas the pull one (Sources, Co-sinks) provide data. This idea has recently been taken advantage of to bring efficient array programming facilities to functional programming (**???**).

This concept is central in the literature on Girard's linear logic (**??**). However, in the case of streams, this idea dates back at least to **?** (**?** gives a good summary of Jacksons' insight).

Our contribution is to bring this idea to stream programming in Haskell. (While duality was used for Haskell array programming, it has not been taken advantage for stream programming.) 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

? 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 prodiving 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 fasion.

7.3 Iteratees

We consider that the state of the art in Haskell stream processing is embodied by Kiselyov's iteratees **?**.

The type for iteratees can be given the following definitions:

```
data I s m a = Done a | GetC (Maybe s → 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 → m (I el m a)
type Enumeratee elo eli m a =
  I eli m a → 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 **?** 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 → m ()) m
type Producer m e = GenT e m ()
type Consumer m e = e → m ()
type Transducer m1 m2 e1 e2 =
  Producer m1 e1 → 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* (**?**), *conduits* (**?**) and *machines* (**?**). 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 (**??**). 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 in-

ner 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 **?** and **?**:

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

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 (**???**) 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 (**??**). 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 **?**: as far as we know this remains to be investigated.

Another line of development would be to implement language support for the linearity convention, directly in Haskell. There has been many proposals to extend functional languages with linear types (see for example (**?**, Ch. 9) for a survey). These proposals are often rather involved, because they typically support advanced forms of polymorphism allowing to abstract over the linearity of an argument. The linearity convention that we use here calls for no such complexity, therefore we hope it may be enough of a motivation to add simple linear type support in research-grade Haskell compilers.

9. Conclusion

We have cast an new light on the current state of coroutine-based computation in Haskell, which we have done so 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

We gratefully thank Koen Claessen, Atze van der Ploeg and Nicolas Pouillard for feedback on drafts of this paper. The source code for this paper is a literate Haskell file, whose latest version is available at this url: https://gist.github.com/jyp/fadd6e8a2a0aa98ae94d

The paper is typeset using pandoc, lhs2TeX and latex.

References

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

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.

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

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

K. Claessen. Parallel parsing processes. *Journal of Functional Programming*, 14(6):741–757, 2004.

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.

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

A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232, Copenhagen, Denmark, 1993. ACM. ISBN 0-89791-595-X. . URL `http://portal.acm.org/citation.cfm?id=165180.165214`.

J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

G. Gonzalez. The pipes package, 2015. URL `http://hackage.haskell.org/packages/pipes`.

J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989. URL `http://citeseer.ist.psu.edu/hughes84why.html`.

M. A. Jackson. *Principles of Program Design*. Academic Press, Inc., Orlando, FL, USA, 1975. ISBN 0123790506.

M. Kay. You pull, I'll push: on the polarity of pipelines. In *Proceeding of Balisage: The Markup Conference*, 2008. URL `http://www.balisage.net/Proceedings/vol3/html/Kay01/BalisageVol3-Kay01.html`.

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

O. Kiselyov. Iteratees. In *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings*, pages 166–181, 2012b. . URL `http://dx.doi.org/10.1007/978-3-642-29822-6_15`.

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

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

O. Kiselyov, S. L. Peyton Jones, and A. Sabry. Lazy v. yield: Incremental, linear pretty-printing. In *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012.*

Proceedings, pages 190–206, 2012. . URL http://dx.doi.org/10.1007/978-3-642-35182-2_14.

E. A. Kmett, R. Bjarnason, and J. Cough. The machines package, 2015. URL https://github.com/ekmett/machines/.

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

K. Mazurak, J. Zhao, and S. Zdancewic. Lightweight linear types in system f. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, pages 77–88. ACM, 2010.

P.-A. Melliès and N. Tabareau. Resource modalities in tensor logic. *Ann. Pure Appl. Logic*, 161(5):632–653, 2010. . URL http://dx.doi.org/10.1016/j.apal.2009.07.018.

G. Munch-Maccagnoni. Formulae-as-types for an involutive negation. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 70:1–70:10, 2014. . URL http://doi.acm.org/10.1145/2603088.2603156.

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

M. Snoyman. The conduit package, 2015. URL http://www.stackage.org/package/conduit.

J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional Programming*, pages 124–132, Pittsburg PA, USA, 2002. ACM. . URL http://portal.acm.org/citation.cfm?id=581491.

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

J. A. Tov. *Practical Programming with Substructural Types*. PhD thesis, Northeastern University, 2012.

P. Wadler. Propositions as sessions. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional Programming*, ICFP '12, pages 273–286, New York, NY, USA, 2012. ACM.

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