Lazy v. Yield: Incremental, Linear Pretty-Printing

Conference Paper · December 2012

DOI: 10.1007/978-3-642-35182-2_14

CITATIONS
5

READS
115

3 authors:

Oleg Kiselyov
Tohoku University
162 PUBLICATIONS **2,168** CITATIONS
SEE PROFILE

Simon Loftus Peyton Jones
Microsoft
381 PUBLICATIONS **17,679** CITATIONS
SEE PROFILE

Amr Sabry
Indiana University Bloomington
92 PUBLICATIONS **2,671** CITATIONS
SEE PROFILE

Lazy v. Yield:
Incremental, Linear Pretty-printing

Oleg Kiselyov, Simon Peyton-Jones, and Amr Sabry

[1] oleg@okmij.org
[2] simonpj@microsoft.com
[3] sabry@cs.indiana.edu

Abstract. We propose a programming style for incremental stream processing based on typed *simple generators*. It promotes modularity and decoupling of producers and consumers just like lazy evaluation. Simple generators, however, expose the implicit suspension and resumption inherent in lazy evaluation as computational effects, and hence are robust in the presence of other effects. Simple generators let us accurately reason about memory consumption. To substantiate our claims we give a new solution to the notorious pretty-printing problem. Like earlier solutions, it is linear, backtracking-free and with bounded latency. It is also simpler to write and reason about, and is compatible with effects including IO, letting us read the source document from a file, and format it as we read.

1 Introduction

Lazy evaluation is regarded as one of the main reasons why functional programming matters [1]. Lazy evaluation lets us write *producers* and *consumers* separately, whereas the two are inextricably intertwined in a call-by-value language. This separation allows a modular style of programming, in which a variety of producers, consumers, and transformers can readily be "plugged together." Lazy evaluation is also an elegant implementation of a form of coroutines, suspending and resuming computations based on the demand for values, giving us memory-efficient, incremental computation 'for free' [2–4].

Extensive experience in Haskell has, however, exposed severe drawbacks of lazy evaluation, which are especially grievous for stream processing of large amounts of data. Lazy evaluation is fundamentally incompatible with computational effects, can cause fatal memory leaks, and greatly inhibits modular reasoning, especially about termination and space consumption. Seemingly innocuous and justified changes to the code or code compositions may lead to divergence, or explosion in memory consumption. We review and illustrate these drawbacks in more detail in §2.

It is therefore worth investigating alternatives to lazy evaluation, which are nearly as convenient, modular, and conducive to incremental computation, and yet are more robust under composition with computational effects. Since lazy evaluation corresponds to an *implicit* and stylized use of coroutines [5–8], it is

natural to consider an *explicit* stylized use of coroutines. Such alternatives, historically known as *generators* or *iterators*, have indeed been used to structure programs using "pipelines" of producers and consumers [9] following Jackson's principles of program design [10]. They have recently re-emerged in various modern languages such as Ruby, Python, C#, and JavaScript. Although the details differ, such languages offer an operator yield that captures a particular pattern in which a computation is suspended, allowed to communicate with another context, and then resumed.[4]

Survey [11] documents a rich variety of yield operators, whose most general variant is tantamount to full first-class delimited continuations. In this paper we study a much lighter-weight variant of yield, which we call *simple generators*. We make the following specific contributions:

- We introduce *typed simple generators* (§3), offering a library of combinators that support efficient stream producers, transducers, and consumers. Despite their simplicity (implementable on a single linear stack without copying), simple generators are capable to solve a variety of stream-processing problems that would normally require lazy evaluation, and yet compose readily with effects.
- We show that simple generators are expressive enough to efficiently implement a particularly recondite problem: that of bounded-lookahead, linear-time, incremental, pretty-printing (§4). This problem has been widely considered to require full coroutines or 'tying of the knot,' which in turn demands lazy evaluation. Our new solution, derived in §5, is distinguished by modularity and ease and precision of the analysis about latency, time, and especially space. The overall pretty-printing becomes a composition of independently developed, tested, and analyzed components. We can swap the components, for example, replacing the component that traverses an in-memory document with a generator that reads the document from a file. The new structure helped us discover and correct edge cases neglected in previous implementations.
- We give benchmarks in §5.4 to validate our analyses of space and time.
- Although we mostly use Haskell for concreteness and the ease of contrasting yield and lazy evaluation, our approach extends to other languages. Monadic style, needed to work with generators in Haskell, is a model of call-by-value languages. Since monads make effects (sometimes painfully) explicit, they give good guidance to implementing and using generators in strict languages – as we demonstrate for OCaml.

Appendix A of the full paper `http://okmij.org/ftp/continuations/PPYield/yield-pp.pdf` gives the derivation of optimal pretty-printing in all detail. The complete Haskell and OCaml code accompanying the paper is available online in the same directory.

[4] In some other languages such as Java, the interleaving of execution between producers and consumers is achieved with threads or even bytecode post-processing (weaving).

2 The problem with laziness

The real virtue of lazy evaluation, emphasized in Hughes's famous paper [1], is that it supports *modular programming* by allowing the *producer* and *consumer* of a data structure to be separated, and then composed in a variety of ways in "plug-and-play" fashion. Alas, lazy evaluation is also fragile:

- Coroutining via lazy evaluation is incompatible with computational effects. With effects, the evaluation order is significant and cannot be left implicit. Adding effects requires re-writing of the code, and, as in call-by-value languages, tangles data producers and consumers (see §2.1);
- Lazy evaluation allows us to implement *cyclic* data transformers as well as linear ones; this is called 'tying the knot.' A number of elegant Haskell algorithms are written that way, e.g., the famous repmin [3]. Alas, it is particularly hard to reason about the termination of such algorithms, in part because the types do not tell us anything about demand and supply of values (see §2.2).
- Reasoning about space requirements of lazy algorithms is notoriously hard, and non-modular: how much space a function may need depends on the context. One case study [12] reports that it took three weeks to write a prototype genomic data mining application in Haskell, and two months to work around laziness frequently causing heap overflows. More examples are discussed at the end of §3 and in a recent paper by the first author [13].

The rest of the section illustrates these problems and the next one demonstrates how yield answers them.

2.1 Effects

Our running example is typical file processing: to read and print a file expanding all tabs[5]. Ideally, reading file data, tab expansion itself, and writing the result should be separated, so each can be replaced. For example, the programmer may chose between two tab expansion algorithms: the naïve replacement of '\t' with 8 spaces, and the sophisticated one, adding spaces up to the next tab stop (the multiple of 8). In idiomatic, lazy Haskell we start by writing the naïve tabX0 and sophisticated algorithms tabX1 assuming that the input data are all in memory, in a String. We do not worry how data got into memory or how it is written out:

```
tabX0, tabX1 :: String → String
tabX0 []          = []
tabX0 ('\t': rest ) = replicate 8 ' ' ++ tabX0 rest
tabX0 (c:rest )   = c : tabX0 rest

tabX1 = go 0 where
 go pos []          = []
 go pos ('\t': rest ) = let pos' = (pos +8) - pos `mod` 8 in
                        replicate (pos' - pos) ' ' ++ go pos' rest
 go pos (c: rest )  = c : go (if c == '\n' then 0 else pos +1) rest
```

[5] We do *not* assume that the file is made of lines of bounded width.

The sophisticated version keeps a bit of local state, the pos argument, for the current output position. The complete function is as follows:

```
expandFile_lazy :: String → IO ()
expandFile_lazy filename = do h ← openFile filename ReadMode
                              str ← hGetContents h
                              putStr (tabX0 str)
```

It opens the file, reads its contents, calls tabX0 to transform the contents, and writes out the result. It is modular, in the sense that it would be the work of a moment to replace tabX0 with tabX1, or by some other transformation entirely.

But we obviously don't want to read in the entire contents of the file into memory, then transform it, and then write it out. We would prefer to read the file on demand, so to process arbitrarily large files in bounded memory. Haskell's hGetContents function does exactly that, returning a string containing lazy thunks that, when forced (ultimately by putStr), read the file.

This on-demand input/output is called "lazy IO." Although very convenient, it is fraught with problems, including deadlocks and leaks of space, file descriptors and other resources [13]. The problems are apparent already in our code. For example, on file reading error an exception would be raised not in hGetContents but rather at some later indeterminate point in putStr. Moreover, file reading operations interleave uncontrollably with other IO operations, which may be fatal if the input file is actually a pipe. These problems really do arise in practice; the standard response is "you should use strict I/O."[6]

Thus lazy evaluation indeed does not work with effects. If we use effects, we have to re-write our tab-expansion code, for example, as follows:

```
expandFile_strict :: String → IO ()
expandFile_strict filename = do
  h ← openFile filename ReadMode; loop h; hClose h
  where loop h = do done ← hIsEOF h
                    if done then return () else hGetChar h ≫= check ≫ loop h
        check '\t' = putStr (replicate 8 ' ')
        check c    = putStr [c]
```

We now distinguish EOF from other input errors. We explicitly close the file as soon as we do not need it. Alas, the tab expansion, reading, and checking for the end of data are all intertwined. Although we can abstract out processing of a character, we cannot abstract out the processing of the entire stream, or easily replace the naïve tab expansion algorithm with the smart tab expansion: we have to re-write the whole reading-writing loop to thread the output pos. We clearly see what John Hughes meant when saying that strict evaluation entangles consumers and producers and inhibits modularity.

The same considerations would apply to any effectful producer or transformer. Thus, there is a real tension between the modular programming style advocated in "Why functional programming matters" and computational effects.

[6] e.g. http://stackoverflow.com/questions/2981582/haskell-lazy-i-o-and-closing-files.

2.2 Recursive knots

Richard Bird's famous repmin function [3], shown below, has made a compelling case for lazy evaluation. The function takes a tree and returns a new tree of the same shape, with every leaf value replaced by the minimum leaf value of the original tree. The new tree is constructed on-the-fly and the original tree is traversed only *once*:

```
data Tree = Leaf Int | Node Tree Tree
repmin :: Tree → Tree
repmin t = tr where (mn, tr) = walk mn t

walk :: Int → Tree → (Int, Tree)
walk mn (Leaf n)      = (n,              Leaf mn)
walk mn (Node t1 t2) = (n1 `min` n2, Node tr1 tr2)
  where (n1,tr1) = walk mn t1
        (n2,tr2) = walk mn t2
```

The main function walk takes the value to put into leaves and a tree and returns its minimum leaf value and the transformed tree. In repmin we pass the minimum leaf value computed by walk as an argument to walk itself, "tying the knot." Crucial is putting the minimum computation mn into a Leaf without evaluating it. Once the computation is eventually evaluated, the resulting minimum value shows up in all leaves. Lazy evaluation is indispensable for repmin.

Alas, small changes easily make the program diverge. For example, suppose we only wish to replace those leaves whose value is more than twice the minimum. If we replace the Leaf n case in walk with the following:

```
walk mn (Leaf n) | n > 2 * mn = (n, Leaf mn)
                 | otherwise  = (n, Leaf n)
```

we get a divergent repmin. Now walk *does* need the value of mn before finishing the traversal, demanding the value it is yet to produce. We may try one of the following fixes:

```
walk mn (Leaf n) = (n, if  n > 2 * mn then Leaf mn else Leaf n)
walk mn (Leaf n) = (n, Leaf (if  n > 2 * mn then mn else n))
```

We leave it as an exercise to determine which one works. The answer is non-obvious, requiring us to do global dataflow analysis in our head, determining which terms are evaluated when. Even experienced functional programmers can make mistakes, and often confess a lack of complete certainty about whether the program is now right. However elegant, "tying the knot" is fragile. To make matters worse, just imagine needing to modify repmin to print the leaves as they are walked!

3 Yield

We now describe how yield can be used as an alternative to lazy evaluation that is robust and compatible with arbitrary effects, and yet has attractive features of lazy evaluation in untangling producers from consumers and allowing them to be developed separately. Since there are many possible variants of yield [11],

as well as several Haskell libraries based on *iteratees* that are similar to ours, we begin by reviewing some background, and then present our particular design contrasting it others. Finally, we revisit the examples in the previous section using our library.

Although we will be using Haskell to introduce yield, we will rely on monadic style, which is a model of call-by-value languages. The modeling of simple generators in Haskell helps implementing them in other languages, such as OCaml.

3.1 Background

Our design of *simple generators* is inspired by CLU's iterators [14] which are themselves inspired by the generators of Alphard [15]. Generators of Alphard were meant as a compositional abstraction of iteration: "generators abstract over control as functions abstract over operations" [15]. These simple generators can be viewed as asymmetric coroutines, a producer and a consumer, that pass data in one direction only, from the producer to the consumer. Besides unidirectionality, simple generators are further restricted: they can be nested but they cannot run side-by-side (and so cannot solve problems like 'same-fringe').

More generally, generators with varying degrees of expressiveness have spread to many languages. A uniform way to understand the variations in expressiveness is to view the various designs as imposing restrictions on delimited continuations [11]. The most general design gives full first-class delimited continuations. Some other (like that of Javascript) expose the continuation as a first-class object in the form of an *external iterator* but restrict the continuation to be a *one-shot* continuation. Even more restricted designs such as Ruby's never expose the continuation and only provide *internal iterators* that can only be used in the context of a foreach-like loop. One simple generators are also restricted in that sense: they never expose the continuation and restrict the implicit continuation to be *one-shot*. The restriction enables a simple and efficient implementation of simple generators on a single linear stack without copying [14].

Haskell's Hackage has a package `generator` for a very simple version of Python generators. It is inefficient, relying on the full delimited continuation monad, and, mainly, does not offer stream transducers.

Like generators, iteratees [13] provide incremental, compositional input processing and a sound alternative to lazy IO. Whereas generators focus on production of values, iteratees are designed around consumption [16]. There are many implementations of iteratees in many languages: just Haskell Hackage has the libraries `iteratee`, `enumerator`, `monad-coroutine`, `iterIO`, `pipes`, `conduit` and even broad categories 'Enumerator' and 'Conduit'. Underlying all these iteratee implementations is the resumption monad, which is tantamount to first-class, multi-shot delimited continuations. Compared to simple generators, iteratees are thus more expressive but much more heavier-weight. They cannot be implemented on linear stack without copying or building auxiliary data structures. The remarkable implementation simplicity and efficiency of simple generators strongly motivates investigating and pushing the limits of their expressiveness.

```
-- Simple generators
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

yield     :: Monad m ⇒ e → Producer m e
runGenT :: Monad m ⇒ Producer m e → Consumer m e → m ()
foldG     :: Monad m ⇒ (s → e → m s) → s → Producer (StateT s m) e → m s

newtype ReaderT env m a = ReaderT { runReaderT :: env → m a }
ask       :: ReaderT env m r

newtype StateT s m a = StateT { runStateT :: s → m (a, s) }
```

Fig. 1. The interface of our Haskell yield library. For completeness, we also include the ReaderT and StateT types from the `transformers` library.

```
type 'a gen          = unit → 'a
type producer        = unit gen
type consumer        = exn → unit
type 'a transducer = 'a gen → 'a gen

val yield     : exn → unit
val iterate : producer → consumer → unit
val foldG    : ('s → exn → 's) → 's → producer → (unit → 's)
```

Fig. 2. The interface of our OCaml library of simple generators. A yielded value is encapsulated in an exception object.

3.2 Producers, Consumers, Transducers

Our design is lightweight: other than type abbreviations, it consists of just three functions. It is summarized in the top portion of Fig. 1 and discussed in detail in the remainder of this section. For comparison, Fig. 2 shows the corresponding interface in OCaml.

The users of simple generators may regard GenT e m as an abstract monad: the type GenT e m a is that of computations that emit values of type e and eventually return a value of type a. The alias Producer emphasizes this interpretation. The concrete type of GenT e m reveals that a producer is structured as an environment (Reader) monad over an arbitrary m. The consumer is stored in an *environment* that the generator can query, or "ask." The consumer thus acts as a *loop body* that the producer invokes to process the 'emitted' element.

With yield being the only primitive producer, and GenT e m being a monad, we may write more complex producers, e.g., emitting characters read from a file:

```
fileGen :: MonadIO m ⇒ Producer m Char
fileGen = do h ←liftIO $ openFile "/tmp/testf.txt" ReadMode
             loop h; liftIO $ hClose h
  where loop h = do done ←liftIO $ hIsEOF h
                    if done then return ()
```

```
                    else liftIO (hGetChar h) >>= yield >> loop h
```

The standard System.IO function putChar is a sample consumer. We hook the producer and putChar by simply saying fileGen `runGenT`putChar, which prints characters as they are read.

The type of yield is an instance of Consumer m a where m is GenT a m'. Therefore, we can build consumers that transform the received element emitting (producing) the result – in short, act as *stream transducers*. Since yield is a consumer that immediately emits the consumed element, runGenT gen yield is the same as gen, and \gen → runGenT gen yield is the identity transducer. Here is a version of the naïve tab expander tabX0 from §2.1 expressed as a transducer:

```
tabY0 :: Monad m ⇒ Transducer (GenT String m) m Char String
tabY0 gen = runGenT gen con
 where con '\t' = yield (replicate 8 ' ')
       con c    = yield [c]
```

Thus tabY0 fileGen is a producer that reads the file and tabifies it, and which can be combined with a consumer like putStr using runGenT.

We may regard Producer e m as an effectful analogue of [e], representing a sequence of elements whose production may require an effect m. Transducers are hence analogues of list transformers, and the list combinators map, fold, etc. have natural counterparts on generators. The following transducer is a map-like one:

```
mapG :: Monad m ⇒ (e1 → e2) → Transducer (GenT e2 m) m e1 e2
mapG f gen = runGenT gen (yield ∘ f)
```

Since transducers are Producer-to-Producer functions, they can be combined by functional composition – letting us add more stream processing stages by composing-in more transducers. For example, a producer which reads the file, upper-cases and tabifies it is composed as (tabY1 ∘ mapG toUpper) fileGen. The library's implementation in terms of an environment monad guarantees that such whole stream processing happens in constant memory.

So far, we merely rewrote expandFile_strict of §2.1 using our generator library. In contrast to §2.1, we can now replace the naïve tab expansion with a sophisticated one without re-writing the file reader. Recall that the sophisticated tab expansion has local state, the current output position. Since our library is parametrized over a monad m, we add local state by merely instantiating m to be the state monad.

```
tabY1 :: Monad m ⇒ Transducer (StateT Int (GenT String m)) m Char String
tabY1 gen = evalStateT (runGenT gen con) 0
 where con e1 = get >>= (\s → lift (f s e1)) >>= put
       f pos '\t' = let pos' = (pos +8) − pos `mod` 8 in
                    yield (replicate (pos' − pos) ' ') >> return pos'
       f pos c = yield [c] >> return (if c == '\n' then 0 else pos +1)
```

To add the sophisticated tab expansion, all we need is to replace tabY0 with tabY1 in the previous code fragments.

The examination of tabY1 points to further abstraction. The internal function f looks like tabX1 in the lazy evaluation example in §2.1, only without the argument rest. Its type fits the pattern s → e → m s of a monadic state trans-

former (with the local state s being Int). We capture the pattern in combinator similar to List.fold:

```
foldG :: Monad m ⇒ (s → e → m s) → s → Producer (StateT s m) e → m s
foldG f s0 gen = execStateT (runGenT gen consumer) s0
 where consumer x = get >>= (\s → lift $ f s x) >>= put
```

We rewrite tabY1 using foldG as:

```
tabY1' = foldG t 0 ≫ return () where
  t pos '\t' = let pos' = (pos +8) − pos `mod` 8 in
               yield (replicate (pos' − pos) ' ') ≫ return pos'
  t pos c = yield [c] ≫ return (if c == '\n' then 0 else pos +1)
```

(we will abbreviate foldG t s ≫ return () as foldG_ t s.)

3.3 Cycles

The function repmin seems out of scope for generators, which are intended for stream processing. The function repmin builds a tree rather than a stream. We can bring repmin into the scope for generators by serializing the resulting tree into a stream of XML-like nodes:

```
data TreeStream = BegNode | EndNode | LeafData Int
serializeX :: Tree → [TreeStream]
```

We now show how to write serializeX ∘ repmin with generators, with the 'side effect' of being able to add arbitrary effects such as debug printing. We start by writing the generator traverse that turns a tree into a producer of TreeStream elements, and collect which collects the elements into a list:

```
traverse :: Monad m ⇒ Tree → Producer m TreeStream
traverse (Leaf i)      = yield (LeafData i)
traverse (Node t1 t2) = do yield BegNode; traverse t1; traverse t2; yield EndNode

collect :: Monad m ⇒ Producer (StateT [e] m) e → m [e]
collect g = foldG (\s e → return $ e:s) [] g >>= return ∘ reverse
```

Then collect ∘ traverse (in the Identity monad) is equivalent to serializeX. We now need to insert a version of repmin that processes the TreeStream elements, replacing LeafData elements by the minimum. This operation clearly needs a look-ahead buffer of type [TreeStream]. The first encountered LeafData node switches on the look-ahead, triggering the accumulation, which continues through the end. We will see a similar buffering in §5.2:

```
repminT gen = foldG go (0,[]) gen >>= \(m,buf) → mapM_ (flush m) (reverse buf)
 where go (m, []) BegNode      = yield BegNode ≫ return (m, [])
       go (m, []) EndNode      = yield EndNode ≫ return (m, [])
       go (m, []) (LeafData x) = return (x,[ LeafData x])
       go (m, b) (LeafData x)  = return (x `min` m,LeafData x : b)
       go (m, b) e             = return (m, e:b)

       flush m (LeafData _) = yield (LeafData m)
       flush _ e            = yield e
```

The serialized repmin-ed tree is then the result of the modular composition collect ∘ repminT ∘ traverse. If we wish to add debug printing, we insert into the cascade a stream transformer that re-emits the elements while printing them.

The stream transformer repminT *obviously* has unbounded look-ahead. Although more difficult to discern, the original repmin also requires unbounded look-ahead, with the tree itself used as an implicit look-ahead 'buffer.'

4 Pretty-printing specification

Oppen [17] defined pretty-printing as a 'nice' formatting of a (tree-structured) document within a fixed page width. The core of his specification of nice formatting – used in all other Haskell implementations – takes documents of the following abstract form:

```
data Doc = Text String | Line | Doc :+: Doc | Group Doc
```

An abstract document is either a string (Text), a potential line break (Line), a composition of two documents side-by-side (:+:), or a *group*. A group specifies a unit whose linebreaks are interpreted consistently. If a group, with all Linebreaks within it interpreted as spaces, fits onto the remainder of the line, the group is formatted this way. Otherwise, the Linebreaks in the group (but not within embedded groups) are treated as newlines. For example, the following simple document:

```
doc1 = Group (Text "A" :+: (Line :+: Group (Text "B" :+: (Line :+: Text "C"))))
```

would be formatted as shown on the left if the width of the page is 5 or more characters, and as shown in the middle if the width of the page is 3 or 4 characters, and as shown on the right if the width of the page is 1 or 2 characters:

```
A B C        A        A
             B C      B
                      C
```

As an executable specification of the problem we take the following inefficient pretty-printer (also used as a starting point by Chitil [18]):

```
type Fit       = Bool          type WidthLeft = Int
type Width     = Int           type PageWidth = Int

pretty1 :: PageWidth → Doc → String
pretty1 w d = fst $ format False w d where
 format :: Fit → WidthLeft → Doc → (String, WidthLeft)
 format f r (Text z)     = (z, r − length z)
 format True  r Line     = (" ", r−1)
 format False r Line     = ("\n",w)
 format f r (d1 :+: d2) = (s1++ s2, r2)
  where (s1,r1) = format f r d1
        (s2,r2) = format f r1 d2
 format f r (Group d)    = format (f || width d ≤ r) r d

 width :: Doc → Width
 width (Text z)     = length z
 width Line         = 1
 width (d1 :+: d2) = width d1 +width d2
 width (Group d)    = width d
```

The function pretty1 is invoked with the width of the page and the document to format. It immediately invokes an interpreter, format, which recursively

traverses and formats the document maintaining a boolean-valued environment variable f and an integer-valued state variable r. The flag f tells if the current document is part of the group that fits on the current line. The flag affects the formatting of Line. The state r tells the remaining available space on the current line; the fit of a group is determined by comparing r with the expected length of the group. This length is calculated by the function width which traverses the document adding up the lengths of its constituent strings.

The executable specification of pretty-printing is clear but greatly inefficient: in the worst case, it takes time exponential in the size n of the input document. The width of an inner group may be repeatedly recomputed, as part of the width computation for ancestor groups as they are being formatted. Furthermore, we cannot begin to format a group until we computed its width; therefore the algorithm has an unbounded look-ahead and its latency is $O(n)$. In contrast, Oppen's original algorithm which is imperative and uses explicit coroutines, is linear in the size of the document, is independent of the page width w to which the document is formatted, and is incremental with a latency bounded by $O(w)$. Attempts to algebraically derive just as efficient Haskell implementation have so far failed [19, 20]. Most Haskell pretty-printing libraries use some form of backtracking and hence cannot have bounded latency. Standing out is Chitil's implementation [18], which matches the classical one in efficiency. It is written however in an iterative style, which amounts to performing a manual continuation-passing style transformation. Swierstra bolstered the case for laziness by showing that a linear, incremental lazy pretty-printing function exists after all [21]. It crucially relies on tying the knot, and its non-divergence is hard to see. Other analyses, in particular estimating space complexity, are difficult as well. The code is complex, with five state parameters, one of which is computed 'backwards.' In fact the solution was developed as an attribute grammar, and hand-translated into Haskell. (Swierstra et al. have since developed mechanical translations and even embedding of attribute grammars in Haskell [4, 22].)

5 Stepwise generation of pretty-printer

We build an efficient pretty-printer by combining two key optimizations: (i) avoiding re-computations of group width by memoization or pre-computation and (ii) pruning, computing the width of a group only as far as needed to determine if the group fits on the line. These optimizations are present in one form or another in all optimal pretty-printing implementations. Our development is distinguished by a systematic, modular and compositional application of the optimizations. We build the pretty-printer as a cascade of separately developed, tested and *analyzed* stream transducers. We stress the ease of analysis and its composability.

Here is a general idea. To avoid re-computing group widths, we may compute the width of all groups beforehand – for example, by traversing the whole document tree and annotating each node with its width. The traversal is standard post-order, linear in the size of the tree. Alas, the annotated tree needs as much

space as the original one. Since we have to traverse all children of the root node to compute its width, we really have to build the whole annotated tree first before we start formatting.

Applying the pruning optimization seems non-obvious, until we make the traversal of the document tree incremental, as a generator of a stream of traversed nodes. The width computation becomes a transducer that adds width annotations to stream elements. The annotated tree is never explicitly constructed. Pruning becomes a straightforward optimization of the group width transducer, bounding its look-ahead. We realize this plan below, step-wise. §5.1 converts the document tree to a stream of nodes, which we then annotate with the horizontal position. §5.2 modifies the annotations so they effectively become group width annotations. §5.3 optimizes the annotation algorithm using pruning. The width-annotated stream is formatted in §5.4. To save space, we focus on the key steps and relegate the details to Appendix A of the full paper.

5.1 Generating document stream

The first step of our plan is converting the document tree to a stream of nodes. The elements of the stream are of the following type:

```
data StreamB = TE String | LE | GBeg | GEnd
```

with constructors for Text and Line and a pair of constructors for entering and leaving a group. The function genB generates a bare stream by in-order traversing the document tree:

```
genB :: Monad m ⇒ Doc → Producer m StreamB
```

Analysis. As genB reaches a text or a line node, it (like traverse in §3) immediately emits the corresponding stream element. Hence genB has unit latency. Since genB is a simple in-order traversal of the tree, the total time to generate the whole stream is linear in the size of the tree. The function needs stack space proportional to the depth d of the tree since genB is not tail-recursive.

We annotate the stream elements with the rolling width, or the horizontal position HP in a hypothetical formatting of the document in a single line:

```
type HP = Int
type HPB = Int
data StreamHPB = TE_b HP String | LE_b HP | GBeg_b HPB | GEnd_b HP
```

All stream elements except GBeg are annotated with the horizontal position at the end of formatting of that element on the hypothetical single line. In particular, GEnd is annotated with the final HP for its group. The node GBeg is however annotated with the horizontal position HPB at the beginning of the formatting of the group. In other words, each node is annotated with the sum of the widths of all preceding nodes including the current. The annotation is done by the simple state transducer trHPB, consuming StreamB and emitting StreamHPB of annotated elements. The horizontal position is the state:

```
trHPB :: Monad m ⇒
  Transducer (StateT HP (GenT StreamHPB m)) m StreamB StreamHPB
```

Analysis. The transforming function merely increments the current horizontal position. It hence does constant amount of work, has unit latency and runs in constant space. The total transformation time is linear in the size of the input stream.

5.2 Determining group widths

The annotated stream is not directly suitable for formatting: when we encounter a group, that is, a GBeg element, we have to decide if the group fits; hence we need the width of the group, or the horizontal position of the group's end. Therefore, we transform StreamHPB into StreamHPA where GBeg will be annotated with final rather than initial HP of the group, that is, the HP of the GEnd element of the group. Clearly this requires look-ahead. Furthermore, since groups can be nested, the look-ahead buffer must be structured, so that we can track several groups in progress:

```
data StreamHPA = TE_a HP String | LE_a HP | GBeg_a HP | GEnd_a HP
type Buffer m = [Buf StreamHPA m]
```

The overall look-ahead Buffer m is a list of simple buffers Buf that each correspond to one unfinished, nested group. A Buf accumulates stream elements corresponding a tree branch, after GBeg and up to and including the matching GEnd. A simple buffer Buf should permit the following operations:

```
buf_empty :: Monad m ⇒ Buf e m
(⊵)       :: Monad m ⇒ Buf e m → e → Buf e m
(⊴)       :: Monad m ⇒ e → Buf e m → Buf e m
buf_ccat  :: Monad m ⇒ Buf e m → Buf e m → Buf e m
buf_emit  :: Monad m ⇒ Buf e m → Producer m e
```

that is, the creation of the empty buffer, appending an element to the buffer b ⊵ e and prepending an element e ⊴ b in constant time, concatenation of two buffers in constant time, and emitting all elements in the buffer in linear time.

The producer of StreamHPA is also a state transducer, from the stream StreamHPB built in the previous section. The state is the look-ahead Buffer m:

```
type St m = StateT (Buffer m) (GenT StreamHPA m)
trHPA :: Monad m ⇒ Transducer (St m) m StreamHPB StreamHPA
trHPA = foldG_ go [] where
 go q (GBeg_b _)       = return (buf_empty:q)
 go (b:q) (GEnd_b p) = pop q (GBeg_a p ⊴ (b ⊵ GEnd_a p))
 go [] (TE_b p)        = yield (TE_a p) ≫ return []   -- ditto for LE
 go (b:q) (TE_b p)     = return ((b ⊵ TE_a p):q)    -- ditto for LE

 pop [] b      = buf_emit b ≫ return []
 pop (b':q) b = return ((buf_ccat b' b):q)
```

GBeg adds a new layer to the Buffer ready to accumulate elements of the new group. Text and Line elements outside of a group are emitted immediately to the output stream. Otherwise, they are accumulated in the Buf of their parent group. GEnd p supplies the final horizontal position p of the group, letting us emit GBeg p and flush the accumulated elements in Buf. Since the terminated group may be part of another, still unfinished group, we delay emitting elements

of the terminated group and put them into the look-ahead buffer of the parent group. Only when the outer group is terminated we finally empty the look-ahead buffer emitting all its elements.

Analysis. Since we cannot emit any group element until we see GEnd, the latency is of the order of n, the size of the whole document (stream). The look-ahead Buffer m is the extra space, again linear in n. Total time is determined by amortization. Assume that each element of the input stream brings us the credit of 2. We spend one credit to yield the element, and to put the element into the buffer (in general, for any constant amount of work within go). Thus all elements in the buffer have one credit left, enough to pay for the linear-time operation buf_emit. Thus, the total time complexity is linear in n.

Hooking up the stream StreamHPA to a linear-time constant-space formatter (similar to the one in §5.4 below) gives the overall pretty-printer, with linear-time complexity but unbounded, $O(n)$ latency and the corresponding amount of extra space. To bound the look-ahead we apply the second optimization below, pruning.

5.3 Pruning

We have just seen that determining the width of each group is expensive since we have to scan the whole group first. However, the exact group width is not necessary: if the width is greater than the page width, we do not need to know by how much. We introduce an 'approximate horizontal position' HPP:

```
data HPP = Small HP | TooFar
data StreamHPP = TE_p HP String | LE_p HP | GBeg_p HPP | GEnd_p HP
```

to use instead of the exact final horizontal position HP to annotate GBeg elements with. GBeg is annotated with TooFar if the final horizontal position of the group is farther than the page width w away from the group's initial horizontal position. Computing HPP requires only bounded, by w, look-ahead. The stream transformer trHPP below is the pruned version of trHPA of the previous section.

The look-ahead BufferP, like the look-ahead Buffer of trHPA, is a sequence of simple Bufs that accumulate delayed elements following a GBeg up to and including the corresponding GEnd. We need to efficiently access the sequence from both ends however; the simple list no longer suffices. The Haskell basis library provides the data structure Seq with the needed algorithmic properties (we import Data.Sequence as S):

```
type BufferP m = (HPL, S.Seq (HPL, Buf StreamHPP m))
bufferP_empty = (0,S.empty)
type HPL = Int
```

If HP is the beginning horizontal position of the group, HPL is a w-offset position: any position after HPL is TooFar. For each accumulated group we compute HPL and make it easily accessible. Furthermore, fst BufferP provides the HPL for the outermost group, so we can easily see if the current HP is too far for that group. If so, we can emit GBeg TooFar and empty its look-ahead Buf.

The transformer trHPP of StreamHPB to StreamHPP is the 'pruned' version of trHPA:

```
type St m = StateT (BufferP m) (GenT StreamHPP m)
trHPP :: Monad m ⇒ PageWidth → Transducer (St m) m StreamHPB StreamHPP
trHPP w = foldG_ go bufferP_empty where
 ...
 check :: BufferP m → HP → GenT StreamHPP m ()
 check (p0,q) p | p ≤ p0 && S.length q ≤ w = return (p0,q)
 check (_,q) p | (_,b) :< q' ← S.viewl q     =
   buf_emit (GBeg_p TooFar ◁ b) ≫ check' q' p
 check' q p | (p',_) :< _ ← S.viewl q = check (p',q) p
            | otherwise               = return bufferP_empty
```

Except for check, it is essentially the same as trHPA of §5.2. The function check prunes the look-ahead: it checks to see if the current horizontal position p exceeds p0, the HPL of the outer group. If so, the outer group is wider than w, which lets us immediately emit $\mathsf{GBeg}_p\mathsf{TooFar}$ and the elements accumulated in the outer Buf. The not-yet-terminated inner group may also turn out too wide: we have to recursively check. The function check also prunes the look-ahead BufferP when it becomes deeper than w, which may happen in the edge case of a document:

```
Group (Group (Group ... :+: Group ...) :+: Group (Group ... :+: Group ...))
```

whose StreamHPB includes an arbitrarily long sequence of GBeg p with the same initial group position p. The first pruning criterion will not be triggered then. In genB of §A.1 we have ensured that each group is at least one character-wide, with no group as a sole child. Therefore, a group that contains at least w descendant groups must be wider than w. Incidentally, this edge case has not been accounted for in [21]; the latter algorithm would need to add yet another state parameter to the formatting function.

Analysis. All S.Seq operations used in the code – adding to the left ($\triangleleft$) or to the right ($\triangleright$) end of the queue and deconstructing the left or the right end with S.viewl or S.viewr – take constant amortized time. Therefore, the analysis of trHPP is similar to the analysis of trHPA, modulo the fact that the total size of the look-ahead BufferP is bounded by w. Therefore, latency and the extra space for the look-ahead buffer are bounded by the page width. The total processing time remains linear in the size of the input stream.

5.4 Putting it all together and benchmarking

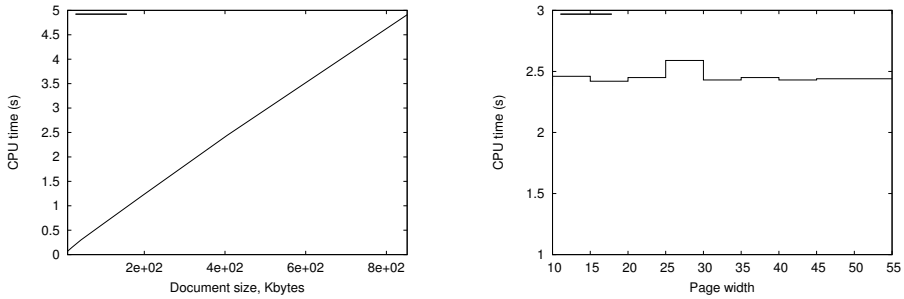
The final step of pretty-printing is the formatting: transforming the pruned StreamHPP to a stream of Strings. To format stream elements LE as spaces or newlines, the formatter keeps track of an indicator if the current group and its ancestors fit on the remainder of the line. The formatter trFormat is straightforward, with unit latency and the overall linear running time, operating in constant space. The complete pretty-printer of a document is a cascade of the width estimators and the formatter, applied to the initial stream generator:

```
pp :: Monad m ⇒ PageWidth → Doc → Producer m String
pp w = trFormat w ∘ trHPP w ∘ trHPB ∘ genB
```

Final Analysis. The total latency is the sum of latencies contributed by all transducers, which is bounded by the page width w. Since all transducers process

the whole stream in time linear to the size of the stream, the total running time of the pretty-printer is linear in the source document size. We need extra space: $O(w)$ for the look-ahead BufferP in trHPP and $O(d)$ (where d is the document depth) for the initial generator genB.

To validate the analyses, we ran a benchmark meant to resemble the full binary tree prepared for pretty-printing. The benchmark, rather than writing



the formatted document to a file, accumulates, in the Writer monad, its total size and the number of lines. The benchmark was compiled with GHC -O2 version 7.0.4 and ran on a i386 FreeBSD 2GHz Pentium 4 system. The running times (in seconds) are the medians of five consecutive runs. The figure on the left plots the running time against the size of the formatted output, for the fixed page size $w = 50$. The figure on the right plots the running time of pretty-printing the same benchmark document (output size 414,870 bytes) against the different values of the page width w. The benchmark validates the analyzes: the running time is a linear function of the document length, independent of w.

6 Conclusions

We have described simple generators to complement or supplant lazy evaluation in stream-processing programs. Like lazy evaluation, simple generators promote modularity, stepwise development and incremental testing by decoupling stream producers, consumers and transformers. Unlike lazy evaluation, simple generators are compatible with effects including IO, and allow modular, composable reasoning about time, latency, and especially about space consumption. We have implemented simple generators as libraries in Haskell and OCaml. The Haskell monadic implementation guides implementations in other languages, making connections with the visitor pattern and dynamic binding clear. In future work, we will formalize the yield calculus and formally relate with call-by-need.

We have illustrated simple generators and demonstrated their expressive power by solving the challenging efficient pretty-printing problem. Our implementation is a new and unexpected solution: efficient pretty-printing was believed to require full delimited continuations or coroutines, which simple generators do not provide. Like the other optimal solutions, it is linear in the size of the input document and has bounded latency. Our solution however was assembled from separately developed and tested components. We have also analyzed time

and space complexity component-by-component, combining the analyses at the end. Our precise analyses discovered previously overlooked edge cases.

Acknowledgments. We thank S. Doaitse Swierstra for helpful discussions. Many helpful suggestions by anonymous reviewers are gratefully acknowledged.

References

[1] Hughes, J.: Why functional programming matters. Comput. J. **32**(2) (April 1989) 98–107
[2] McIlroy, M.D.: Power series, power serious. J. Funct. Program. **9**(3) (May 1999) 325–337
[3] Bird, R.S.: Using circular programs to eliminate multiple traversals of data. Acta Informatica **21** (1984) 239–250 10.1007/BF00264249.
[4] Viera, M., Swierstra, S.D., Swierstra, W.: Attribute grammars fly first-class: how to do aspect oriented programming in Haskell. In: ICFP. (2009) 245–256
[5] Henderson, P., Morris, Jr., J.H.: A lazy evaluator. In: POPL, New York, NY, USA, ACM (1976) 95–103
[6] Ariola, Z.M., Maraist, J., Odersky, M., Felleisen, M., Wadler, P.: A call-by-need lambda calculus. In: POPL, New York, NY, USA, ACM (1995) 233–246
[7] Garcia, R., Lumsdaine, A., Sabry, A.: Lazy evaluation and delimited control. In: POPL, New York, NY, USA, ACM (2009) 153–164
[8] Chang, S., Van Horn, D., Felleisen, M.: Evaluating call-by-need on the control stack. In: Trends in functional programming, Springer (2011) 1–15
[9] Kay, M.: You pull, I'll push: on the polarity of pipelines. In: Proc. Balisage: The Markup Conference. Volume 3 of Balisage Series on Markup Technologies. (2009)
[10] Jackson, M.A.: Principles of Program Design. Academic Press, Inc., Orlando, FL, USA (1975)
[11] James, R.P., Sabry, A.: Yield: Mainstream deliminted continuations. In: Theory and Practice of Delimited Continuations. (2011)
[12] Clare, A., King, R.D.: Data mining the yeast genome in a lazy functional language. In: Practical Aspects of Declarative Languages. Volume 2562 of LNCS. Springer-Verlag (2003) 19–36
[13] Kiselyov, O.: Iteratees. In: FLOPS. Volume 7294 of LNCS., Springer (2012) 166–181
[14] Liskov, B.: A history of CLU. Technical Report 561, MIT LCS (April 1992)
[15] Shaw, M., Wulf, W.A., London, R.L.: Abstraction and verification in Alphard: defining and specifying iteration and generators. Comm. ACM **20**(8) (1977) 553–564
[16] Lato, J.W.: Understandings of iteratees. `http://johnlato.blogspot.com/2012/06/understandings-of-iteratees.html` (11 June 2012)
[17] Oppen, D.C.: Prettyprinting. ACM Trans. Program. Lang. Syst. **2**(4) (October 1980) 465–483
[18] Chitil, O.: Pretty printing with lazy dequeues. ACM Trans. Program. Lang. Syst. **27**(1) (January 2005) 163–184
[19] Hughes, J.: The design of a pretty-printing library. In: Advanced Functional Programming, First Int. Spring School, Springer-Verlag (1995) 53–96
[20] Wadler, P.: A prettier printer. In: The Fun of Programming. A Symposium in honour of Professor Richard Bird's 60th birthday, Oxford (2003)

[21] Swierstra, S.D.: Linear, online, functional pretty printing (corrected and extended version). Technical Report UU-CS-2004-025a, Utrecht University (2004)
[22] Dijkstra, A.: Stepping through Haskell. PhD thesis, Utrecht University, Department of Information and Computing Sciences (2005)

A Detailed stepwise generation of pretty-printer

This section fills in the outline in §5 of the stepwise development of the optimal pretty-printer, and gives the complete details. §A.1 converts the document tree to a stream of nodes, which we annotate with the horizontal position in §A.2. §A.3 modifies the annotations so they effectively become group width annotations. §A.4 optimizes the annotation algorithm using pruning. The width-annotated stream is formatted in §A.5 and (more optimally) §A.6.

A.1 Generating document stream

The first step of our plan is converting the document tree to a stream of nodes, to be transformed and eventually formatted. The elements of the stream are of the following type:

```
data E ab a = TE a String | LE a | GBeg ab | GEnd a
type StreamB = E () ()
```

with the variants for a Text and Line element and for the entering and leaving a group. The development in the main body of the paper, §5, defined a sequence of closely-related data types StreamB, StreamHPB, etc. To avoid the need to constantly re-define TE, LE and similar constructors, we introduce a single data type E and suitably parametrize it by the annotations ab and a to cover the whole range of data types in §5, from StreamB to StreamHPP. In the latter, GBeg had a different annotation (an estimate of the horizontal position) from the annotations on the other elements. StreamB, as before, describes an annotation-free, bare stream.

The function genB generates a bare stream by in-order traversing the document tree. The function is quite like traverse already explained in §3:

```
genB :: Monad m ⇒ Doc → GenT StreamB m ()
genB (Text "")    = return ()
genB (Text z)     = yield (TE () z)
genB Line         = yield (LE ())
genB (d1 :+: d2) = genB d1 ≫ genB d2
genB (Group d0) | Just d ← norm d0 =
  yield (GBeg ()) ≫ genB d ≫ yield (GEnd ())
 where
 norm (Group d)        = norm d
 norm (Text "")        = Nothing
 norm (Text "" :+: d) = norm d
 norm d                = Just d
genB _ = return ()
```

If the document were on a disk, in a sort of XML file, genB could be understood as reading the file and emitting the nodes as they are parsed. The function genB performs normalization of the document, ensuring that: (i) every group is strictly wider than any of its children groups (thus eliminating Group (Group ...)); (ii) any group is at least one-character wide. Normalization is often overlooked, yet critical: without it no pretty-printing algorithm can have bounded look-ahead. For example, in a document with a branch (Group ∘ Group ∘ ... ∘ Group) (Text "")

with an arbitrarily long sequence of Groups, any pretty-printing algorithm without normalization has to scan the whole branch, which can be arbitrarily long, to determine that it does not contribute to the width of the current branch. We discuss this bad edge-case in more detail in §A.4.

Analysis. As genB reaches a text or a line node (whose content is reflected in the formatted output), it immediately emits the corresponding stream element. Hence genB has unit latency. Since genB is a simple in-order traversal of the tree, the total time to generate the whole stream is linear in the size n of the tree. The function needs stack space proportional to the depth d of the tree since genB is not tail-recursive (and neither were formatting functions in §4).

A.2 Computing the horizontal position

We now annotate the stream elements with the rolling width, or the horizontal position HP in a hypothetical formatting of the document in a single long line:

```
type HP = Int
type HPB = Int

type StreamHPB = E HPB HP
```

All stream elements except GBeg are annotated with the horizontal position at the end of formatting of that element on the hypothetical single line. In particular, GEnd is annotated with final HP for its group. The node GBeg is however annotated with the horizontal position HPB at the beginning of the formatting of the group. In other words, each node is annotated with the sum of the widths of all preceding nodes including the current. The annotation is done by the simple state transducer, consuming StreamB and emitting StreamHPB of annotated elements. The horizontal position is the state:

```
trHPB :: Monad m ⇒
  GenT StreamB (StateT HP (GenT StreamHPB m)) ()
  → GenT StreamHPB m ()
trHPB = foldG_ go 0
 where
 go :: Monad m ⇒ HP → StreamB → GenT StreamHPB m HP
 go p (TE _ z) = let p' = p +length z in
                 yield (TE p' z) ≫ return p'
 go p (LE _)   = let p' = p +1 in
                 yield (LE p')   ≫ return p'
 go p (GBeg _) = yield (GBeg p) ≫ return p
 go p (GEnd _) = yield (GEnd p) ≫ return p
```

Here is the annotated stream generated for our sample document doc1 from §4:

```
Generated: GBeg 0
Generated: TE 1 "A"
Generated: LE 2
Generated: GBeg 2
Generated: TE 3 "B"
Generated: LE 4
Generated: TE 5 "C"
```

```
Generated: GEnd 5
Generated: GEnd 5
```

The last HP is 5, at which the string "C" and both groups end.

Analysis. The transforming function go does constant amount of work. Therefore, trHPB has unit latency; total transformation time is linear in the size of the input stream. The function trHPB works in constant space.

A.3 Determining group widths

The annotated stream is not directly suitable for formatting: when we encounter a group, that is, a GBeg element, we have to decide if the group fits; hence we need the width of the group, or the horizontal position of the group's end. Therefore, we transform StreamHPB into StreamHPA where GBeg will be annotated with final rather than initial HP of the group, that is, the HP of the GEnd element of the group. Clearly this requires look-ahead. Furthermore, since groups can be nested, the look-ahead buffer must be structured, so we can track several groups in progress:

```
type StreamHPA = E HP HP
type Buffer  m = [Buf StreamHPA m]
```

The overall look-ahead Buffer m is a list of simple buffers Buf that each correspond to one unfinished, nested group. A Buf accumulates stream elements corresponding a tree branch, after GBeg and up to and including the matching GEnd.

The simple buffer Buf should permit the following operations:

```
buf_empty :: Monad m ⇒ Buf e m
(⊵)       :: Monad m ⇒ Buf e m → e → Buf e m
(⊴)       :: Monad m ⇒ e → Buf e m → Buf e m
buf_ccat  :: Monad m ⇒ Buf e m → Buf e m → Buf e m
buf_emit  :: Monad m ⇒ Buf e m → GenT e m ()
```

that is, the creation of the empty buffer, appending an element to the buffer b ⊵ e and prepending an element e ⊴ b in constant time, concatenation of two buffers in constant time, and emitting all elements in the buffer in linear time. Essentially, Buf is a tree, which buf_emit traverses. We chose the following implementation for Buf:

```
type Buf e m = GenT e m ()
buf_empty = return ()
b ⊵ e = b ≫ yield e
e ⊴ b = yield e ≫ b
buf_ccat b1 b2 = b1 ≫ b2
buf_emit b = b
```

One may substitute any other implementation: All code below accesses the buffer only through the abstract interface.

The producer of StreamHPA is also a state transducer, from the stream StreamHPB built in the previous section. The look-ahead Buffer m is the state:

```
trHPA :: Monad m ⇒
```

```
  GenT StreamHPB
       (StateT (Buffer  m) (GenT StreamHPA m)) ()
  → GenT StreamHPA m ()
trHPA = foldG_ go []
 where
 go q (GBeg _)        = return (buf_empty:q)
 go (b:q) e@(GEnd p) = pop q (GBeg p ◁ (b ▷e))
 go []  e     = yield  e ≫ return  []
 go (b:q) e = return ((b ▷  e): q)

 pop []  b      = buf_emit b ≫ return  []
 pop (b': q) b = return ((buf_ccat  b'  b): q)
```

GBeg adds a new layer to the Buffer ready to accumulate elements of the new group. Text and Line elements outside of a group are emitted immediately to the output stream. Otherwise, the are accumulated in the Buf of their parent group. GEnd p brings us the final horizontal position p of the group, letting us emit GBeg p and flush the accumulated elements in the Buf. Since the terminated group may be part of another, still unfinished group, we delay the emitting elements of the terminated group and put them into the look-ahead buffer of the parent group. Only when the outer group is terminated we finally empty the look-ahead buffer emitting all its elements. The following code, reflecting the sequence of transformations so far:

```
runGenT (trHPA ∘ trHPB ∘ genB $ doc1)
            (\i  → putStrLn $  "Generated:␣" ++ show i)
```

prints StreamHPA for the sample document doc1:

```
Generated: GBeg 5
Generated: TE 1 "A"
Generated: LE 2
Generated: GBeg 5
Generated: TE 3 "B"
Generated: LE 4
Generated: TE 5 "C"
Generated: GEnd 5
Generated: GEnd 5
```

Compared with streamHPB in §A.2, GBeg tells the final position of its group. We stress the incremental development our algorithm: after one part of the overall transformation is developed, we may immediately test it, and analyze it.

Analysis. Since we cannot emit any group element until we see GEnd, the latency is of the order of n, the size of the whole document (stream). The look-ahead Buffer m is the extra space, again linear in n. Total time is determined by amortization. Assume that each element of the input stream brings us the credit of 2. We spend one credit to yield the element, and to put the element into the buffer (in general, for any constant amount of work within go). Thus all elements in the buffer have one credit left, enough to pay for the linear-time operation buf_emit. Thus, the total time complexity is linear in n. Furthermore, we built the new algorithm in incremental fashion, without all tied up in knots. Not only can we clearly see non-divergence, we are also able to accurately and easily estimate time and space complexities.

If we hook up the stream StreamHPA to a linear-time constant-space formatter (similar to the one in §A.6 below), we obtain the overall pretty-printer, with linear-time complexity but unbounded, $O(n)$ latency, taking the corresponding amount of extra space. To bound the look-ahead we apply the second optimization, pruning.

A.4 Pruning

We have just seen that determining the width of each group is expensive since we have to scan the whole group first. However, the exact group width is not necessary: if the width is greater than the page width, we do not need to know by how much. We introduce an 'approximate horizontal position' HPP:

```
data HPP = Small HP | TooFar deriving Show

type StreamHPP = E HPP HP
```

to use instead of the exact final horizontal position HP to annotate GBeg elements with. GBeg is annotated with TooFar if the final horizontal position of the group is farther than the page width w away from the group's initial horizontal position. Computing HPP requires only bounded, by w, look-ahead. The stream transformer trHPP described in this section is the optimized, pruned, version of trHPA of the previous section.

The look-ahead BufferP, like the look-ahead Buffer of trHPA, is a sequence of simple Bufs that accumulate delayed elements following a GBeg up to and including the corresponding GEnd. We will need to efficiently access the sequence from both ends, however; the simple list no longer suffices. The Haskell basis library provides the data structure Seq with the needed algorithmic properties (we import Data.Sequence as S):

```
type BufferP m = (HPL, S.Seq (HPL, Buf StreamHPP m))
bufferP_empty = (0,S.empty)
type HPL = Int
```

If HP is the beginning horizontal position of the group, HPL is a w-offset position: any position after HPL is TooFar. For each accumulated group we compute HPL and make it easily accessible. Furthermore, fst BufferP provides HPL for the outermost group, so we can easily see if the current HP is too far for that group. If so, we can emit GBeg TooFar and empty its look-ahead Buf.

The transformer trHPP of StreamHPB to StreamHPP is the 'pruned' version of trHPA (we also add debug printing to show the trace of the consumed stream). It is the state transducer with the state BufferP:

```
trHPP :: (Monad m, MonadIO m) ⇒
  PageWidth
  → GenT StreamHPB
          (StateT (BufferP m) (GenT StreamHPP m)) ()
  → GenT StreamHPP m ()
trHPP w = foldG_ go' bufferP_empty
 where
 go' s e = do
  liftIO ∘ putStrLn $ "trHPP: read: " ++ show e
```

```
 go s e

go b@(_,q) (TE p z) | S.null q =
  yield (TE p z) ≫ return b
go b@(_,q) (LE p)   | S.null q =
  yield (LE p)   ≫ return b
go b@(_,q) (GEnd p) | S.null q =
  yield (GEnd p) ≫ return b

go (_, q) (GBeg p) | S.null q =
  return (p+w,S.singleton (p+w,buf_empty))
go (p0,q) (GBeg p) =
  check (p0,q ▷ (p+w,buf_empty)) p
go (p0,q) (GEnd p) | q' :> (_,b) ← S.viewr q =
  pop p0 q' (GBeg (Small p) ◁ (b ▷ (GEnd p)))

go (p0,q) (TE p z) = check (p0,push (TE p z) q) p
go (p0,q) (LE p)   = check (p0,push (LE p) q) p

push e q | q' :> (p,b) ← S.viewr q =
  q' ▷ (p,b ▷ e)

pop p0 q b | S.null q =
  buf_emit b ≫ return bufferP_empty
pop p0 q b | q' :> (p,b') ← S.viewr q =
  return (p0, q' ▷ (p,buf_ccat b' b))

check (p0,q) p | p ≤ p0 && S.length q ≤ w =
  return (p0,q)
check (_,q) p | (_,b) :< q' ← S.viewl q =
  buf_emit (GBeg TooFar ◁ b) ≫ check' q' p

check' q p | (p', _) :< _ ← S.viewl q = check (p',q) p
           | otherwise                = return bufferP_empty
```

It is the longest function in our code. Except for check, it is essentially the same as trHPA of §A.3. When no accumulation takes place (BufferP has no Bufs), input stream elements are immediately relayed to the output stream. Receiving GBeg switches on the accumulation. We compute HPL for that group as p+w and push a new Buf to BufferP to accumulate the elements to be later emitted for that group. GEnd p terminates the accumulation for the group. The accumulated elements, preceded by GBeg (Small p), are added to the buffer of the parent group or emitted, see pop.

The new, compared to trHPA, function check prunes the look-ahead: it checks to see if the current horizontal position p exceeds p0, the HPL of the outer group. If so, the outer group is wider than w, which lets us immediately emit GBeg TooFar and the elements accumulated in the outer Buf. The not-yet-terminated inner group may also turn out too wide: we have to recursively check.

The function check also prunes the look-ahead BufferP when it becomes deeper than w, which may happen in the edge case of a document like the following:

```
Group (
```

```
Group (Group ...  :+: Group ...)  :+:
Group (Group ...  :+: Group ...))
```

whose StreamHPB includes an arbitrarily long sequence of GBeg p with the same initial group position p. The first pruning criterion will not be triggered then. Recall that genB of §A.1 has ensured that each group is at least one character-wide, with no group as a sole child. Therefore, a group that contains at least w descendant groups must be wider than w. Incidentally, this edge case has not been accounted for in [21]; the latter algorithm would need to add yet another state parameter to the formatting function. The clarity of our algorithm helped us discover the edge case.

The sample pruned StreamHPP produced by trHPP for the page width 3 is as follows:

```
trHPP: read: GBeg 0
trHPP: read: TE 1 "A"
trHPP: read: LE 2
trHPP: read: GBeg 2
trHPP: read: TE 3 "B"
trHPP: read: LE 4
Generated: GBeg TooFar
Generated: TE 1 "A"
Generated: LE 2
trHPP: read: TE 5 "C"
trHPP: read: GEnd 5
Generated: GBeg (Small 5)
Generated: TE 3 "B"
Generated: LE 4
Generated: TE 5 "C"
Generated: GEnd 5
trHPP: read: GEnd 5
Generated: GEnd 5
```

The listing contains the debugging messages printed as soon as an element of the input StreamHPB is received. The receiving and emitting of stream elements clearly interleave. When the looked-ahead group elements TE "A", LE, TE "B", LE tell that the group must be already wider than the page width 3, we indeed emit GBeg TooFar right away, without waiting for GEnd.

Analysis. All S.Seq operations used in the code – adding to the left ($\triangleleft$) or to the right ($\triangleright$) end of the queue and deconstructing the left or the right end with S.viewl or S.viewr – take constant amortized time. Therefore, the analysis of trHPP is similar to the analysis of trHPA, modulo the fact that the total size of the look-ahead BufferP is bounded by w. Therefore, latency and the extra space for the look-ahead buffer are bounded by the page width. The total processing time remains linear in the size of the input stream.

A.5 Formatting

The final step of pretty-printing is the formatting: transforming the pruned StreamHPP to a stream of Strings. To format stream elements TL as spaces

or newlines, the formatter keeps track of the boolean indicator if the current group fits on the remainder of the line. Since groups nest, we maintain the stack of boolean indicators:

```
type Fits = [Bool]
```

The formatter is the state transducer with the state (Fits,HPL). The HPL part of the state is the horizontal position at the end of the current line. If the final position of a group is over that HPL, the group does not fit. The group certainly does not fit if its final position is TooFar:

```
trFormat0 :: Monad m ⇒
   PageWidth
   → GenT StreamHPP (StateT (Fits, HPL) (GenT String m)) ()
   → GenT String m ()
trFormat0 w = foldG_ go ([False], w)
 where
 go (f, l) (TE _ z) = yield z ≫ return (f, l)
 go (f@(True:_), l) (LE _) = yield "␣" ≫ return (f, l)
 go (f@(False:_), l) (LE p) = yield "\n" ≫ return (f, p+w)
 go (f, l) (GBeg TooFar) = return (False:f, l)
 go (f, l) (GBeg (Small p)) = return ((p ≤ l):f, l)
 go (_:f, l) (GEnd _) = return (f, l)
```

The result of formatting the sample document for different page widths matches the expected one, shown in §4.

Analysis. Since we immediately emit the received data, latency is unit. Overall processing time is linear in the size of the stream. Alas, we need extra space for the stack Fits, which we eliminate next.

A.6 Better Formatting

We exploit a particular form of the stack Fits of indicators if a group and its ancestor groups fit within the rest of the current line. Clearly if a group fits, all of its children groups fit as well. Therefore, if an element of the Fits stack is False, the elements below must all be False. Conversely, if an element is True, the elements above it, through the top of the stack, must all be True. Thus the whole stack is adequately represented by a single number, the number of True elements at the top:

```
type FitI = Int
```

The optimized formatting transducer is as follows:

```
trFormat :: Monad m ⇒
  PageWidth
  → GenT StreamHPP (StateT (FitI, HPL) (GenT String m)) ()
  → GenT String m ()
trFormat w = foldG_ go (0, w)
 where
 go (f, l) (TE _ z) = yield z ≫ return (f, l)
 go (0, l) (LE p) = yield "\n" ≫ return (0, p+w)
 go (f, l) (LE _) = yield "␣" ≫ return (f, l)
 go (0, l) (GBeg TooFar) = return (0, l)
```

```
go (0, l) (GBeg (Small p)) = return
                               (if p ≤ l then 1 else 0, l)
go (f, l) (GBeg _)         = return (f+1,l)
go (0, l) (GEnd _)         = return (0, l)
go (f, l) (GEnd _)         = return (f−1,l)
```

Analysis. Like the earlier trFormat0, the formatter clearly has unit latency and the overall linear running time. It now operates in constant space.