Lazy v. Yield
Incremental, Linear Pretty-printing

Oleg Kiselyov   Simon Peyton-Jones   Amr Sabry

APLAS 2012
Kyoto, Japan   Dec 12, 2012

Incremental stream processing, pervasive in practice, makes the best case for lazy evaluation. Lazy evaluation promotes modularity, letting us glue together separately developed stream producers, consumers and transformers. Lazy list processing has become a cardinal feature of Haskell. It also brings the worst in lazy evaluation: its incompatibility with effects and unpredictable and often extraordinary use of memory. Much of the Haskell programming lore are the ways to get around lazy evaluation.

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 and latency. The remarkable implementation simplicity and efficiency of simple generators strongly motivates investigating and pushing the limits of their expressiveness.

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 modular, structured as a cascade of separately developed stream transducers, which makes it simpler to write, test and to precisely analyze latency, time and space consumption. It is compatible with effects including IO, letting us read the source document from a file, and format it as we read.

Outline

1. The splendors and miseries of lazy evaluation
   - Generators bring modularity to strict languages
   - Generators are compatible with effects
   - Typed *simple generators*
2. Non-trivial example of simple generators
   - A new solution to the pretty-printing problem
   - Surprising power of very simple generators
   - Reasoning and accurate prediction of latency, time and space complexities

The talk, and the paper, have two main parts. The first is about the universally acknowledged main attraction of lazy evaluation, and the universally acknowledged main drawback. Generators bring the main attraction – modularity and compositionality – to strict, call-by-value languages, and they are compatible with effects. This talk is about a simple version of generators. Some may say too simple, since they are limited in expressiveness. But they are very simple to implement. What we can still do with these too simple generators? Well, more than we thought.

The second part is a non-trivial illustration of simple generators: a new solution to the pretty-printing problem – a contribution by itself. It is quite a surprising contribution since some experts didn't think simple generators are up to this job. The example shows how can we reason about and accurately predict the time and space performance of a program. You have heard it right: we will be reasoning about *space*, in *Haskell*. And it is *simple*, once you stay within the framework.

Outline

1. The splendors and miseries of lazy evaluation
   - Generators bring modularity to strict languages
   - Generators are compatible with effects
   - Typed *simple generators*
2. Non-trivial example of simple generators
   - A new solution to the pretty-printing problem
   - Surprising power of very simple generators
   - Reasoning and accurate prediction of latency, time and space complexities

The approach is cross-platform (Haskell, OCaml)
We will be using Haskell

Simple generators are not language-specific, and we have the implementation of generators in Haskell and OCaml. In this talk we will use Haskell.

Lazy Evaluation

- ○ Modularity, reuse, decoupling consumers and producers: why functional programming matters
- × Incompatible with effects
- × Reasoning about space is excruciating

Furthermore:

- Tying the knot (around yourself)
- AI search problems are all but impossible
- Debugging is difficult

Let us first recall the acknowledged, greatest attractions and greatest drawbacks of Lazy evaluation – something that both Lennart Augustsson and Bob Harper publicly agree upon (and this doesn't happen often). John Hughes in his famous paper regarded lazy evaluation as one of the main reasons why functional programming matters. It is because it decouples consumers and producers of data and enables reuse, permitting and encouraging programming by composing combinators. We'll see examples next. Alas, this greatest asset is incompatible with effects, and we pay for it with the excruciating difficulty of estimating the space requirements of a program and plugging memory leaks.

There are further points like tying the knot – a fascinating application on one hand, which also lets us tie up ourselves with our own rope. Lazy evaluation makes it all but impossible to program search in large spaces, where laziness, or memoization, is exactly the wrong trade-off. There is no time to talk about them here.

Lazy Evaluation: modularity

```
any :: (a → Bool) → [a] → Bool
any f = or ∘ map f
```

Lennart Augustsson: More points for lazy evaluation, May 2011.

So, modularity. We will be using the main point example from Lennart Augustsson's well-discussed blog post from last year. The example is the function `any`, which tells if an element of a list satisfies a given predicate. The function can be defined by the composition of two already written functions `or` and `map`.

Lazy Evaluation: modularity

```
any :: (a → Bool) → [a] → Bool
any f = or ∘ map f

t1 = any (>10) [1..]
-- True
```

Lennart Augustsson: More points for lazy evaluation, May 2011.

Further, `any` stops as soon as it finds the satisfying element. To demonstrate, we use it with an infinite list, obtaining the result `True`. We will not get any result in a strict language: `or` does not get to work before `map f` finishes – which, if it is applied to an infinite list, does not.

Lazy Evaluation: modularity

```
any :: (a → Bool) → [a] → Bool
any f = or ∘ map f

t1 = any (>10) [1..]
-- True

t2 = any (>10) ∘ map read ∘ lines $ "2\n3\n5\n7\n11\n13\nINVALID'
-- True
```

Lennart Augustsson: More points for lazy evaluation, May 2011.

We can grow the chain farther. For example, the input list of numbers could be the result of parsing a column of text. Splitting into lines, parsing, comparing – all happens incrementally and on demand. Indeed, t2 returns True even though "INVALID" is unparseable as a number. The evaluation has really stopped before the entire list is consumed.

Lazy evaluation and effects

Naive reading

```
(any (>10) ∘ map read ∘ lines) `fmap`
    (read_string  fname :: IO String )
```

× Lost incremental processing

That was greatly appealing. But what if the string to parse and search is to be read from a file? Suppose read_string is an IO action that reads a string from a file. But how much to read? Only the pure pipeline in the parenthesis can tell, but it is evaluated after the reading action finishes. (IO) actions can't run half-way, return a part of the result, and then resume. Since read_string gets no feedback on how much to read, it has to read the whole file. Hence we lost the incrementality and the early termination, right after the satisfying element is found.

Lazy evaluation and effects

Naive reading

```
(any (>10) ∘ map read ∘ lines) `fmap`
   (read_string  fname :: IO String)
```

╳ Lost incremental processing

Lazy IO

```
t2_lazy  = do
  h    ← openFile  test_file    ReadMode
  str  ← hGetContents h
  let  result   = any (>10) ∘ map read ∘ lines $  str
  return  result
```

╳ Lost sanity

But we can read on demand – there is Lazy IO, like in the code below. The library function hGetContents arranges for such an incremental read, as more of the string is demanded.

Lazy IO has *many* problems, and I have already talked about them, not so while ago and not so far from here. I just mention one problem: when the handle h is closed? When the garbage collector gets around to it, if ever. The lazy IO code thus leaks a scarce resource (the file handle). Lazy IO is especially problematic when the input comes from a pipe or a network channel. It is not specified how much hGetContents really reads. It usually reads by buffer-full and it may read-ahead arbitrarily, which for communication channels can (and does) lead to a deadlock.

Lazy evaluation and space complexity

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.

Amanda Clare and Ross D. King: Data Mining the Yeast Genome in a Lazy Functional Language, PADL2003.

As to reasoning about space complexity, let me refer to a case study presented as PADL2003. Since then, the GHC strictness analyzer got better, and memory got much cheaper. Still the point stands, and we shall see the example.

If not lazy, then strict?

```
t2_strict  =
  bracket (openFile test_file  ReadMode) hClose (loop (>10))
 where
  loop f h = do
    r ← liftM f $ liftM read $ hGetLine h
    if r then return True else loop f h
```

◯ No file handle leaks

╳ Lost compositionality

On the plus side, the handle is definitely closed, immediately right after the result is obtained. But we have lost the compositionality: we have to program the recursion explicitly, essentially inlining `or`. See Augustsson's post for more discussion why we have to do this. The accompanying code also discusses monadic lists (streams) and demonstrates why streams leak resources like a file handle.

If neither lazy nor strict, then what?

Non-Strict languages

- ○ modularity and reuse
- × incompatible with effects
- × any space cost model at all?

Strict languages (CBV)

- ○ good time and space cost model
- ○ compatible with effects
- × difficult modularity and reuse:
  coupling data producers and consumers

Let's recap. Non-strict languages promote modularity and compositionality by decoupling producers and consumers. They are incompatible with effects. One may wonder if non-strict languages have any space cost model at all, in practical programs.

Strict languages, in contrast, have the good space cost model, are compatible with effects, but couple consumers and producers of data and hence break modularity and inhibit reuse. So, we should try to repair CBV by finding a way to uncouple the producers and the consumers. And that's exactly what generators do.

Typed Simple Generators

```
type GenT e m
instance MonadTrans (GenT e)

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
```

The interface is simple: one abstract `GenT e m` and three functions. `GenT e` is a monad transformer for generators that yield the value `e` in some monad `m`. The three type aliases meant to clarify the meaning. See the paper for discussion of and pointers to the variety of generators and how simple generators fit in.

Generating

```
fileG :: (GBracket m, MonadIO m) ⇒ FilePath → Producer m Char
fileG fname =
  gbracket (liftIO $ openFile fname ReadMode) (liftIO ∘ hClose) loop
 where
 loop h = do
  b ← liftIO $ hIsEOF h
  if b then return () else liftIO (hGetChar h) ≫= yield ≫ loop h
```

- No file handle leaks
- Mere producer: no coupled consumers

The code looks like the strict IO code we saw earlier. The handle is closed as soon as the result (or an exception) are obtained. But we don't process any read data here: we only *yield* them. This is a mere producer, written with no consumers in sight. This is the first illustration how generators uncouple the production and consumption of data.

Consuming

```
orG :: Monad m ⇒ Producer (ErrT Bool m) Bool → m Bool
orG gen = either id (const False) `liftM` runErrT (runGenT gen orC)
 where
  orC :: MonadError Bool m ⇒ Consumer m Bool
  orC True  = throwError True
  orC False = return ()
```

Transforming

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

```
type TrState m = StateT String (GenT String m)
linesG  :: Monad m ⇒ Transducer (TrState m) m Char String
linesG  gen = foldG tr [] gen >>= at_end
 where
 tr  s '\n' = yield (reverse s) >> return []
 tr  s c    = return (c:s)
 at_end []  = return ()
 at_end s   = yield (reverse s)
```

The slide shows a stateless transformer and a state transformer, converting one generator into another. In particular, linesG converts the generator of characters into the generator of lines, by accumulating characters until it sees a newline, at which point the accumulated string is yielded.

Composing

```
anyG f = orG ∘ mapG f

t2_gen :: IO Bool
t2_gen = anyG (>10) ∘ mapG read ∘ linesG $ fileG test_file
```

- Functional composition and reuse
- No file handle leaks

Finally we compose the separately written producers, consumers and transformers. The code here looks quite like the lazy code – but we do read from a file and close the file handle promptly.

Simplicity of Simple Generators

simple generators: asymmetric, one-shot, implicit supensions

× Cannot run side-by-side

○ Very easy to implement:
single linear stack without copying

- dynamic binding (common terms)
- resumable exception (CL terms)
- environment, Reader monad (Haskell terms)

```
type GenT e m                        = ReaderT (e → m ()) m
```

Like lazy evaluation, generators are stylized co-routines. Although some versions of generators are as powerful as general, first-class suspensions (delimited control), our simple generators are quite restricted. They are asymmetric, one-shot, implicit supensions.Our simple generators are indeed implemented via the Reader monad. Here is what GenT really looks like.

Pretty-printing problem

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

results in . . .

$w = 5$	$w = 3$ or 4	$w = 1$ or 2
A B C	A B C	A B C

Requirements

- time linear in document size n
- independent of the line width w
- bounded latency
- constant space (when read from a file)

Latency is the amount of time from the moment a piece of data such as "A" is seen by the formatter till the moment it is placed in the output document.

Possible algorithms

- Compute the size of a group; if fits, format Line as spaces
  $t = O(2^n), \quad \ell$ unbounded
- Backtracking
  $t = O(2^n), \quad s = O(n), \quad \ell$ unbounded
- Pre-compute group size
  $t = O(n), \quad s = O(n), \quad \ell$ unbounded

One may imagine several algorithms. The simplest one computes the size of a group; if fits, formats Line as spaces. Alas, it takes, in the worst case, the exponential time with respect to the document size n and has unbounded latency. Backtracking, common to many practical libraries, is even worse, since we need to store the non-committed output. Pre-computing group sizes is better, but it takes space of the order of document size n. Again, the latency is unbounded.

Modular pretty-printing algorithm

$\boxed{\text{Generating}}$	$t = \Theta(n)$	$s = \Theta(\log n)$	$\ell = 1$
$\boxed{\text{Annotating}}$	$t = \Theta(n)$	$s = \Theta(1)$	$\ell = 1$
$\boxed{\text{Estimating}}$	$t = \Theta(n)$	$s = \Theta(n)$	$\ell = \Theta(n)$
$\boxed{\text{Formatting}}$	$t = \Theta(n)$	$s = \Theta(1)$	$\ell = 1$
Total	$t = \Theta(n)$	$s = \Theta(n)$	$\ell = \Theta(n)$

data Stream = TE String | LE | GBeg | GEnd

This is (the first phase) of our algorithm, developed incrementally. The overall algorithm: functional composition of all the stages. The result is a stream of strings. We add the latencies ℓ and take the maximum of time t and memory s. The overall result is unsatisfactory: latency and memory linear in the document size.

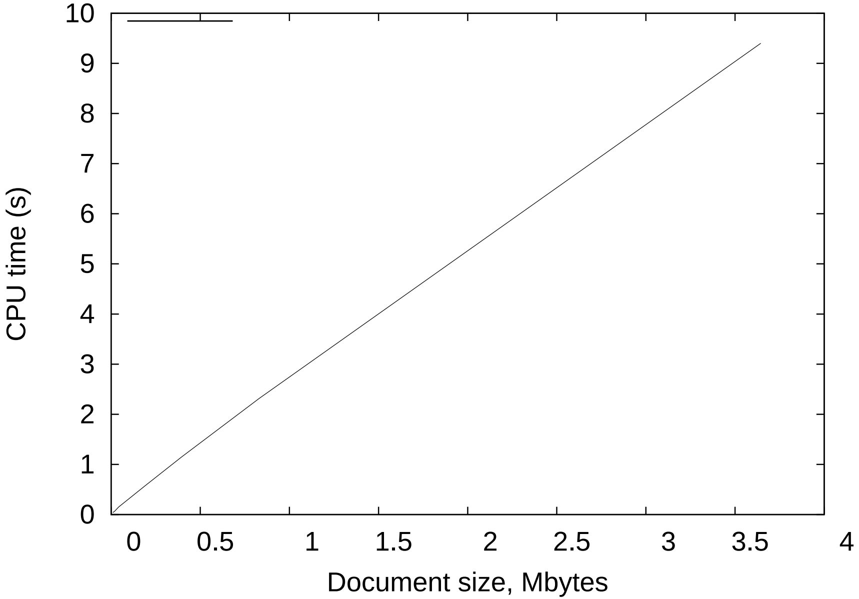
Modular pretty-printing algorithm

$\boxed{\text{Generating}}$	$t = \Theta(n)$	$s = \Theta(\log n)$	$\ell = 1$
$\boxed{\text{Annotating}}$	$t = \Theta(n)$	$s = \Theta(1)$	$\ell = 1$
$\boxed{\text{Pruning}}$	$t = \Theta(n)$	$s = \Theta(w)$	$\ell = \Theta(w)$
$\boxed{\text{Formatting}}$	$t = \Theta(n)$	$s = \Theta(1)$	$\ell = 1$
Total	$t = \Theta(n)$	$s = \Theta(w)$	$\ell = \Theta(w)$

data Stream = TE String | LE | GBeg | GEnd

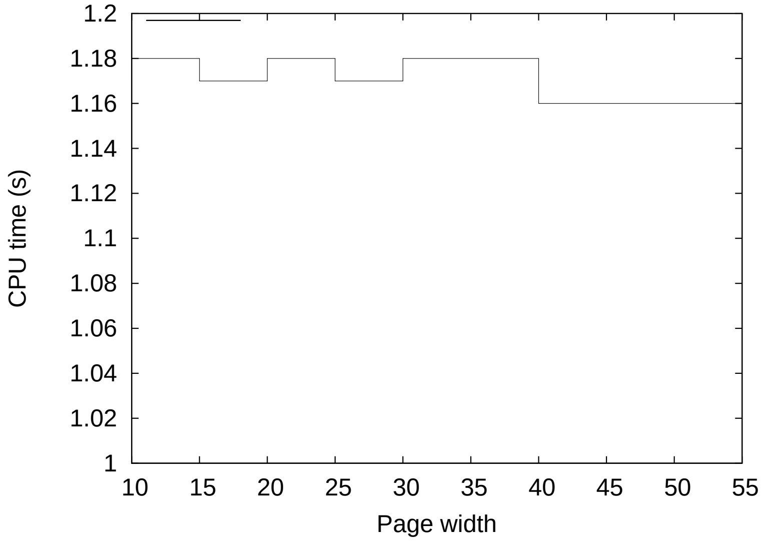
But we make a small change, replacing the group size estimation with pruning: we only need to know the group size to the extent it is less than w. With this change, we obtain the algorithm with the desired complexity. Please see the full paper for the complete details.

Benchmark: time vs document size



It's time for benchmarks. We've just seen the analysis, which was performed on the source code, in Haskell. Let us see how those predictions bear out in practice. The full description of the benchmark is in the paper, and the code is on the web. GHC 7.4.2. Here is the first benchmark, the running time (reported by the GHC RTS statistics) in seconds vs. the document size in MBytes. The page width is constant, 50. Recall the analysis predicted the linear relationship. I am embarrassed to say that the observation is really a straight line through the origin. You can run the benchmark yourself and see for yourselves.

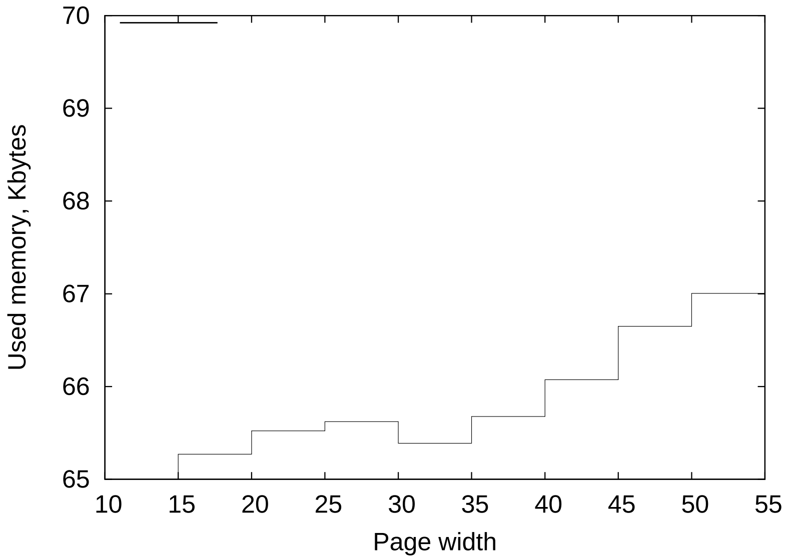
Benchmark: time vs page width



Constant document size 405Kb

We have also predicted that the running time of formatting does *not* depend on the page width. Here is what we see in practice: the running time of the benchmark in seconds vs the page width, for a constant document size 405Kb.

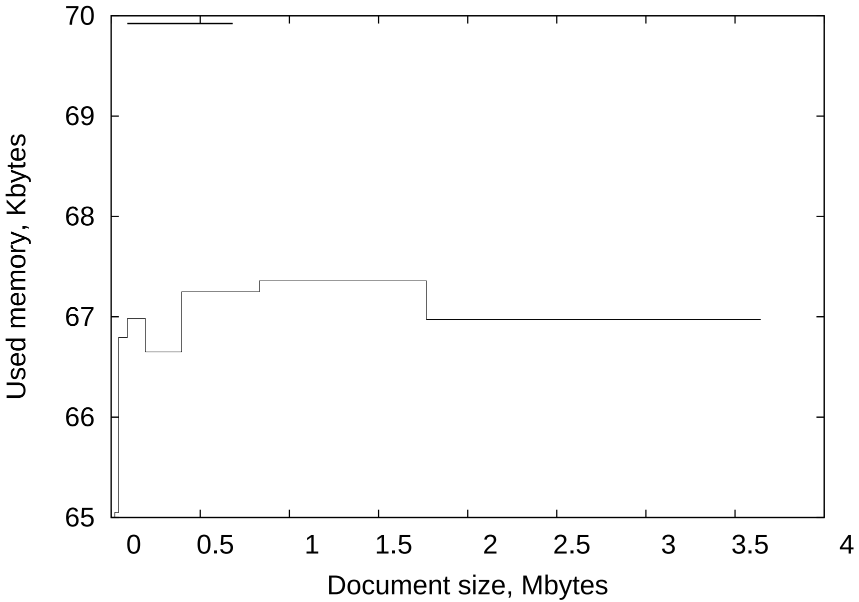
Benchmark: memory vs page width



Constant document size 405Kb

Space benchmarks are more interesting. Space is much more difficult to reason about in Haskell. We see the average resident memory in KBytes, as reported by the GC statistics, vs page width. Recall our analysis predicted $\Theta(w)$.

Benchmark: memory vs document size



Finally, the most interesting benchmark: memory needed for formatting vs the document size. Our code analysis said the formatting should run in constant memory for the constant page width, 50.

I must stress that the document width (the X axis) is in *megabytes*, and the memory used is in *kilobytes*, and the spread over the Y axis is only 5K.

Gotcha

```
data Counts = Counts Int Int deriving  Show

instance Monoid Counts where
  mempty = Counts 0 0
  mappend (Counts c1 l1) (Counts c2 l2)  = Counts (c1+ c2) (l1+ l2)
```

Initially I didn’t have strict fields below, and the memory usage was linear! I looked at the code and found a leak, *outside* of the formatting code. Instead of writing the formatted data into a file, the becnhmark, to avoid the complexities of IO, counted the total number of characters and the total number of newlines. The problem was in this counting code, shown on the slide.

Anyone sees a memory leak here?

Gotcha

```
data Counts = Counts !Int !Int deriving Show

instance Monoid Counts where
  mempty = Counts 0 0
  mappend (Counts c1 l1) (Counts c2 l2) = Counts (c1+ c2) (l1+ l2)
```

It is in the addition. Within the main formatting, the monadic style makes the dependencies plain and GHC can strictify things appropriately. With generators, we have build the walled guarded, from which laziness is exorcised. But once we venture outside, laziness is lurking to get you.

Conclusions

Simple generators to complement or supplant lazy evaluation in stream-processing programs

A new and unexpected solution to the pretty-printing problem:

- no coroutines
- modular development, modular analysis

- reuse and compositionality in the presence of effects
- reasoning about space complexity

Let us recount the lessons.

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.

We implenmented the generators in OCaml and Haskell.

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. Simplicity can be a virtue.