Streaming for tail-recursive programs

Jeremy Gibbons
University of Oxford
WG2.1#60, May 2005

1. Metamorphisms

```
> meta :: (b->Maybe (c,b)) -> (b->a->b) -> b -> [a] -> [c]
> meta f g b as = unfoldr f (foldl g b as)

> foldl :: (b -> a -> b) -> b -> [a] -> b
> foldl f e []     = e
> foldl f e (a:as) = foldl f (f e a) as

> unfoldr :: (b -> Maybe (c,b)) -> b -> [c]
> unfoldr f b = case f b of
>   Just (c, b') -> c : unfoldr f b'
>   Nothing      -> []
```

2. Streaming

```
> stream :: (b->Maybe (c,b)) -> (b->a->b) -> b -> [a] -> [c]
> stream f g b as =
>   case f b of
>     Just (c, b') -> c : stream f g b' as
>     Nothing      ->
>       case as of
>         (a:as') -> stream f g (g b a) as'
>         []      -> []
```

3. Streaming condition

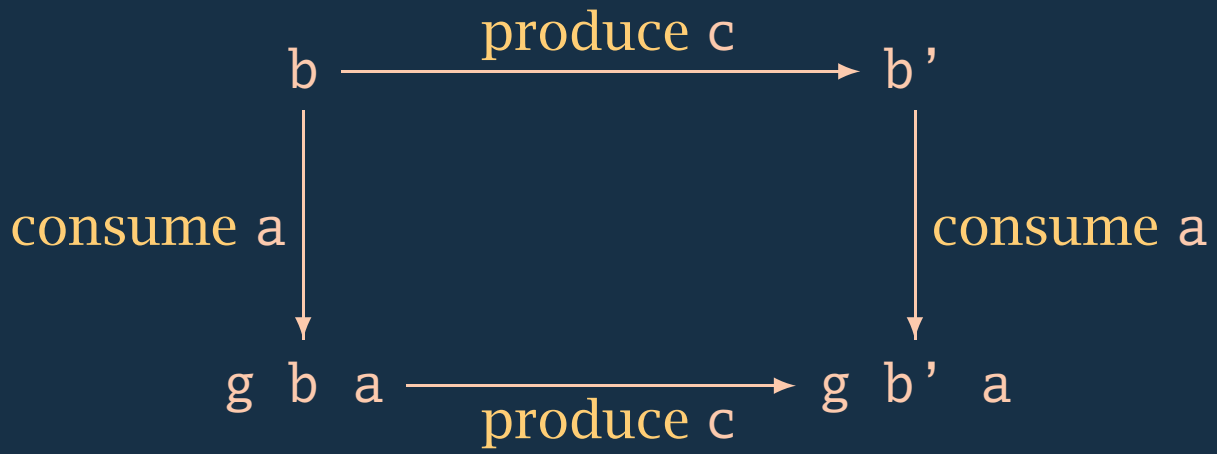
The *streaming condition* for `f` and `g` is that whenever

```
f b = Just (c,b')
```

then, for any `a`,

```
f (g b a) = Just (c, g b' a)
```

It's a kind of invariant property.



4. Streaming theorem

Theorem: if the *streaming condition* holds for `f` and `g`, then

```
stream f g b as = unfoldr f (foldl g b as)
```

for all *finite* lists `as`.

For example, the streaming condition holds for `unCons` and `(++)`, and so the composition

```
unfoldr unCons . foldl (++) []
```

can be streamed.

5. Generalizing

As stated above, streaming is about converting lists to lists.

It is not too difficult to generalize the output to other datatypes.
(But I have no convincing examples...)

How about the input? `foldl` does not generalize easily to other datatypes.
(Yes, I know about Alberto Pardo's work, but that doesn't seem to help here.)

I now think streaming is really about *tail recursion*.
`foldl` is just one example—perhaps the most familiar—of tail recursion.

6. The tail recursion pattern

Here is a (fairly) general form of tail recursion.

```
> tr :: (a->Maybe a) -> (b->a->b) -> b -> a -> b
> tr h g b a =
>   case h a of
>     Nothing -> b
>     Just a' -> tr h g (g b a) a'
```

For example,

```
> reverse = tr safetail conshead []
>   where safetail []     = Nothing
>         safetail (x:xs) = Just xs
>         conshead ys (x:xs) = x:ys
```

(WLOG, we assume that the final value of type a is not used.)

7. Refactoring tail recursion

I claim:

```
tr h g b a = foldl g b (trace h a)
```

where

```
> trace :: (a->Maybe a) -> a -> [a]
> trace h a = case h a of
>   Just a' -> a : trace h a'
>   Nothing -> []
```

Thus, any tail-recursive program is a `foldl` after a `trace`.

Tail-recursive programs are necessarily linearly recursive, and that is where the lists in streaming come from.

8. Streaming a tail recursion

```
> trstream :: (b->Maybe (c,b)) -> (b->a->b) -> (a->Maybe a) ->
>             b -> a -> [c]
> trstream f g h b a =
>   case f b of
>     Just (c, b') -> c : trstream f g h b' a
>     Nothing      ->
>       case h a of
>         Just a' -> trstream f g h (g b a) a'
>         Nothing -> []
```

Provided the streaming condition holds for `f` and `g`,

```
trstream f g h b a = unfoldr f (foldl g b (trace h a))
```

on as for which `trace h a` is finite.

9. Applications?

I feel there is a connection with Doaitse's *Polish Parsers* (ICFP 2004). One aspect of that work is to make parsing *online*: the output is a list of revelations, either bits of AST or confirmations of inputs consumed, and this output is *streamed*.

I'd welcome other suggestions!