Streaming algorithms (extended abstract)

Jeremy Gibbons, Oxford University Computing Laboratory

March 2004

Introduction

We are interested in capturing and studying *recurring patterns of computation*, such as 'folds' (consuming data structures — technically, witnesses to the initiality of a datatype) and unfolds (generating data structures — witnesses to finality). As has been strongly argued by the recently popular *design patterns* movement, identifying and exploring such patterns has many benefits: reuse of abstractions, rendering 'folk knowledge' in a more accessible format, providing a common vocabulary of discourse, and so on. What distinguishes patterns in functional programming from patterns in object-oriented and other programming paradigms is the better 'glue' available, which allows the patterns to be expressed as *abstractions within the language*, rather than having to resort to extra-linguistic prose and diagrams.

It is natural to consider also compositions of these operations. *Hylomorphisms*, consisting of a fold after an unfold, have been well-studied; informally, the unfold generates an intermediate 'virtual data structure', which is immediately consumed by the fold. In this paper, we consider the opposite composition: a fold consumes an input data structure to construct some intermediate (possibly unstructured) data, which is unfolded into an output data structure. Note that the two data structures may now be of different shapes, since they do not meet. Indeed, such programs may often be thought of as *representation changers*, converting from one structured representation of some abstract data to a different structured representation. We use the term *metamorphism* for such compositions, as they 'metamorphose' data structures.

Hylomorphisms enjoy an almost-automatic deforestation property: assuming only certain rather weak strictness conditions, the intermediate data structure may be bypassed altogether and the two phases fused. In general, metamorphisms enjoy no such property. Nevertheless, sometimes fusion is possible; under certain conditions, some of the unfolding may be performed before all of the folding is complete. This kind of fusion can be helpful for controlling the size of the intermediate data. Perhaps more importantly, it can allow conversions between infinite data representations. For this reason, we call such fused metamorphisms *streaming algorithms*; they are the main subject of this paper. We encountered them fortuitously while trying to describe some data compression algorithms, but have since realized that they are an interesting construction in their own right.

This paper is an extended abstract of a fuller version (to appear in *Mathematics of Program Construction*, 2004).

Streaming

We consider metamorphisms of the form $\mathsf{unfoldr}\, f \cdot \mathsf{foldl}\, g\, c$. Essentially the problem is a matter of finding some kind of *invariant* of the state of the foldl that determines the initial behaviour of the unfold. This idea is captured by the following definition.

Definition 1 *The* streaming condition *for f and g is:*

$$f\ c = \mathsf{Just}\,(b, c') \quad \Rightarrow \quad f\ (g\ c\ a) = \mathsf{Just}\,(b, g\ c'\ a)$$

for all a, b, c and c'.

Informally, the streaming condition states the following: if c is a state from which the unfold would produce some output element (rather than merely the empty list), then so is the modified state $g\ c\ a$ for any a;

moreover, the element b output from c is the same as that output from $g\ c\ a$, and the residual states c' and $g\ c'\ a$ stand in the same relation as the starting states c and $g\ c\ a$. In other words, 'the next output produced' is invariant under consuming another input.

This invariant property is sufficient for the unfold and the fold to be fused into a single process, which alternates between consuming inputs and producing outputs. We define:

$$\begin{array}{lcl}
\mathit{stream} & :: & (\gamma \to \mathsf{Maybe}\,(\beta,\gamma)) \to (\gamma \to \alpha \to \gamma) \to \gamma \to [\alpha] \to [\beta] \\
\mathit{stream}\ f\ g\ c\ x & \mathrel{\widehat{=}} & \textbf{case}\ f\ c\ \textbf{of} \\
 & & \quad \mathsf{Just}\,(y,c') \to y : \mathit{stream}\ f\ g\ c'\ x \\
 & & \quad \mathsf{Nothing} \to \textbf{case}\ x\ \textbf{of} \\
 & & \qquad a : x' \to \mathit{stream}\ f\ g\ (g\ c\ a)\ x' \\
 & & \qquad [\,] \to [\,]
\end{array}$$

Informally, $\mathit{stream}\ f\ g :: \gamma \to [\alpha] \to [\beta]$ involves a producer f and a consumer g; maintaining a state c, it consumes an input list x and produces an output list y. If f can produce an output element b from the state c, this output is delivered and the state revised accordingly. If f cannot, but there is an input a left, it is consumed and the state revised accordingly. When the state is wrung dry and the input is exhausted, the process terminates.

Formally, the relationship between the metamorphism and the streaming algorithm is given by the following theorem.

Theorem 1 (Streaming Theorem) *If the streaming condition holds for f and g, then*

$$\mathit{stream}\ f\ g\ c\ x \quad = \quad \mathsf{unfoldr}\ f\ (\mathsf{foldl}\ g\ c\ x)$$

on finite lists x.

Note that the result relates behaviours on finite lists only: on infinite lists, the foldl never yields a result, so the metamorphism may not either, whereas the streaming process can be productive — indeed, that is the whole point of introducing streaming.

Flushing streams

Sometimes the streaming condition fails to hold, even though streaming is possible. This is a common situation with streaming algorithms: the producer function needs to be more cautious when interleaved with consumption steps than it does when all the input has been consumed. In the latter situation, there are no further inputs to invalidate a commitment made to an output; but in the former, a subsequent input might invalidate whatever output has been produced. The solution to this problem is to introduce a more sophisticated version of streaming, which proceeds more cautiously while input remains, but switches to the normal more aggressive mode if and when the input is exhausted.

This is formalized in the following generalization of *stream*:

$$\begin{array}{lcl}
\mathit{fstream} :: (\gamma \to \mathsf{Maybe}\,(\gamma,\beta)) \to (\gamma \to \alpha \to \gamma) \to (\gamma \to [\beta]) \to \gamma \to [\alpha] \to [\beta] \\
\mathit{fstream}\ f\ g\ h\ c\ x & \mathrel{\widehat{=}} & \textbf{case}\ f\ c\ \textbf{of} \\
 & & \quad \mathsf{Just}\,(b,c') \to b : \mathit{fstream}\ f\ g\ h\ c'\ x \\
 & & \quad \mathsf{Nothing} \to \textbf{case}\ x\ \textbf{of} \\
 & & \qquad a : x' \to \mathit{fstream}\ f\ g\ h\ (g\ c\ a)\ x' \\
 & & \qquad [\,] \to h\ c
\end{array}$$

The difference between *fstream* and *stream* is that the former has an extra argument, h, a 'flusher'; when the state is wrung as dry as it can be and the input is exhausted, the flusher is applied to squeeze the last few elements out. This is a generalization, because supplying the trivial flusher that always returns the empty list reduces *fstream* to *stream*. (In fact, *fstream* can also be expressed in terms of *stream*, so the two are equally expressive.)

The relationship of metamorphisms to flushing streams is a little more complicated than that to ordinary streams, but nevertheless a similar result can be stated.