Streaming representation-changers

Jeremy Gibbons

Computing Laboratory, University of Oxford
www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons
jeremy.gibbons@comlab.ox.ac.uk

Abstract. *Unfolds* generate data structures, and *folds* consume them. A *hylomorphism* is a fold after an unfold, generating then consuming a *virtual data structure*. A *metamorphism* is the opposite composition, an unfold after a fold; typically, it will convert from one data representation to another. In general, metamorphisms are less interesting than hylomorphisms: there is no automatic *fusion* to *deforest* the intermediate virtual data structure. However, under certain conditions fusion is possible: some of the work of the unfold can be done before all of the work of the fold is complete. This permits *streaming metamorphisms*, and among other things allows conversion of *infinite data representations*. We present a theory of metamorphisms and outline some examples.

1 Introduction

Folds and *unfolds* in functional programming [18, 28, 3] are well-known tools in the programmer's toolbox. Many programs that consume a data structure follow the pattern of a fold; and dually, many that produce a data structure do so as an unfold. In both cases, the structure of the program is determined by the structure of the data it processes.

It is natural to consider also compositions of these operations. Meijer [30] coined the term *hylomorphism* for the composition of a fold after an unfold. The *virtual data structure* [35] produced by the unfold is subsequently consumed by the fold; the structure of that data determines the structure of both its producer and its consumer. Under certain rather weak conditions, the intermediate data structure may be eliminated or *deforested* [39], and the two phases fused into one slightly more efficient one.

In this paper, we consider the opposite composition, of an unfold after a fold. Programs of this form consume an input data structure using a fold, constructing some intermediate (possibly unstructured) data, and from this intermediary produce an output data structure using an unfold. 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. Despite the risk of putting the reader off with yet another neologism of Greek origin, we cannot resist coining the term *metamorphism* for such compositions, because they typically metamorphose representations.

In general, metamorphisms are perhaps less interesting than hylomorphisms, because there is no nearly-automatic deforestation. 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 [4], but have since realized that they are an interesting construction in their own right.

The remainder of this paper is organized as follows. Section 2 summarizes the theory of folds and unfolds. Section 3 introduces metamorphisms, which are unfolds after folds. Section 4 presents a theory of streaming, which is the main topic of the paper. Section 5 provides an extended application of streaming, and Section 6 outlines two other applications described in more detail elsewhere. Finally, Section 7 discusses some ideas for generalizing the currently rather list-oriented theory, and describes related work.

2 Origami programming

We are interested in capturing and studying *recurring patterns of computation*, such as folds and unfolds. As has been strongly argued by the recently popular *design patterns* movement [8], 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 that the better 'glue' available in the former [20] allows the patterns to be expressed as *abstractions within the language*, rather than having to resort to informal prose and diagrams.

We use the notation of Haskell [25], the de facto standard lazy functional programming language, except that we take the liberty to use some typographic effects in formatting, and to elide some awkwardnesses (such as type coercions and qualifications) that are necessary for programming but that obscure the points we are trying to make.

Most of this paper involves the datatype of lists:

$$\mathbf{data}\ [\alpha] = [\,] \mid \alpha : [\alpha]$$

That is, the datatype $[\alpha]$ of lists with elements of type α consists of the empty list $[\,]$, and non-empty lists of the form $a : x$ with head $a :: \alpha$ and tail $x :: [\alpha]$.

The primary patterns of computation over such lists are the *fold*, which consumes a list and produces some value:

$$\begin{array}{ll} \mathsf{foldr} & :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\ \mathsf{foldr}\ f\ b\ [\,] & \mathrel{\widehat{=}}\ b \\ \mathsf{foldr}\ f\ b\ (a : x) & \mathrel{\widehat{=}}\ f\ a\ (\mathsf{foldr}\ f\ b\ x) \end{array}$$

and the *unfold* [16], which produces a list from some seed:

$$\begin{aligned}
&\mathsf{unfoldr} \quad :: (\beta \to \mathsf{Maybe}\,(\alpha, \beta)) \to \beta \to [\alpha] \\
&\mathsf{unfoldr}\, f\ b \mathrel{\widehat{=}} \mathbf{case}\ f\ b\ \mathbf{of} \\
&\qquad\qquad \mathsf{Just}\,(a, b') \to a : \mathsf{unfoldr}\, f\ b' \\
&\qquad\qquad \mathsf{Nothing} \quad\ \to [\,]
\end{aligned}$$

Here, the datatype Maybe augments a type α with an additional value Nothing:

$$\mathbf{data}\ \mathsf{Maybe}\ \alpha = \mathsf{Nothing} \mid \mathsf{Just}\ \alpha$$

The foldr pattern consumes list elements from right to left (following the way lists are constructed); as a variation on this, there is another fold which consumes elements from left to right:

$$\begin{aligned}
&\mathsf{foldl} \quad\ :: (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta \\
&\mathsf{foldl}\, f\ b\,[\,] \quad \mathrel{\widehat{=}} b \\
&\mathsf{foldl}\, f\ b\,(a : x) \mathrel{\widehat{=}} \mathsf{foldl}\, f\,(f\ b\ a)\ x
\end{aligned}$$

We also use the operator scanl, which is like foldl but which returns all partial results instead of just the final one:

$$\begin{aligned}
&\mathsf{scanl} \quad\ :: (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to [\beta] \\
&\mathsf{scanl}\, f\ b\,[\,] \quad \mathrel{\widehat{=}} [b] \\
&\mathsf{scanl}\, f\ b\,(a : x) \mathrel{\widehat{=}} b : \mathsf{scanl}\, f\,(f\ b\ a)\ x
\end{aligned}$$

We introduce also a datatype of internally-labelled binary trees:

$$\mathbf{data}\ \mathsf{Tree}\ \alpha = \mathsf{Node}\,(\mathsf{Maybe}\,(\alpha, \mathsf{Tree}\ \alpha, \mathsf{Tree}\ \alpha))$$

with fold operator

$$\begin{aligned}
&\mathsf{foldt} \qquad\qquad\qquad\qquad :: (\mathsf{Maybe}\,(\alpha, \beta, \beta) \to \beta) \to \mathsf{Tree}\ \alpha \to \beta \\
&\mathsf{foldt}\, f\,(\mathsf{Node}\ \mathsf{Nothing}) \qquad \mathrel{\widehat{=}} f\ \mathsf{Nothing} \\
&\mathsf{foldt}\, f\,(\mathsf{Node}\,(\mathsf{Just}\,(a, t, u))) \mathrel{\widehat{=}} f\,(\mathsf{Just}\,(a, \mathsf{foldt}\, f\ t, \mathsf{foldt}\, f\ u))
\end{aligned}$$

and unfold operator

$$\begin{aligned}
&\mathsf{unfoldt} \quad :: (\beta \to \mathsf{Maybe}\,(\alpha, \beta, \beta)) \to \beta \to \mathsf{Tree}\ \alpha \\
&\mathsf{unfoldt}\, f\ b \mathrel{\widehat{=}} \mathbf{case}\ f\ b\ \mathbf{of} \\
&\qquad\qquad \mathsf{Nothing} \qquad\ \to \mathsf{Node}\ \mathsf{Nothing} \\
&\qquad\qquad \mathsf{Just}\,(a, b_1, b_2) \to \mathsf{Node}\,(\mathsf{Just}\,(a, \mathsf{unfoldt}\, f\ b_1, \mathsf{unfoldt}\, f\ b_2))
\end{aligned}$$

(It would be more elegant to define lists and their recursion patterns in the same style, but for consistency with the Haskell standard prelude we adopt its definitions. We could also condense the above code by using various higher-order combinators, but for accessibility we refrain from doing so.)

The remaining notation will be introduced as it is encountered. For more examples of the use of these and related patterns in functional programming, see [13], and for the theory behind this approach to programming, see [12]; for a slightly different view of both, see [3].

3 Metamorphisms

In this section we present three simple examples of metamorphisms, or representation changers in the form of unfolds after folds. These three represent the entire spectrum of possibilities: it turns out that the first permits streaming automatically (assuming lazy evaluation), the second does so with some work, and the third does not permit it at all.

3.1 Reformatting lines

The classic application of metamorphisms is for dealing with *structure clashes* [22]: data is presented in a format that is inconvenient for a particular kind of processing, so it needs to be rearranged into a more convenient format. For example, a piece of text might be presented in 70-character lines, but required for processing in 60-character lines. Rather than complicate the processing by having to keep track of where in a given 70-character line a virtual 60-character line starts, good practice would be to separate the concerns of rearranging the data and of processing it. A *control-oriented* or *imperative* view of this task can be expressed in terms of coroutines: one coroutine, the processor, repeatedly requests the other, the rearranger, for the next 60-character line. A *data-oriented* or *declarative* view of the same task consists of describing the intermediate data structure, a list of 60-character lines. With lazy evaluation, the two often turn out to be equivalent; but the data-oriented view may be simpler, and is certainly the more natural presentation in functional programming.

We define the following Haskell functions.

$$
\begin{array}{l}
\mathit{reformat} \quad :: \mathit{Integer} \to [[\alpha]] \to [[\alpha]] \\
\mathit{reformat}\ n \quad \widehat{=}\ \mathit{writeLines}\ n \cdot \mathit{readLines} \\
\\
\mathit{readLines} \quad :: [[\alpha]] \to [\alpha] \\
\mathit{readLines} \quad \widehat{=}\ \mathsf{foldr}\ (+\!\!+)\ [\,] \\
\mathit{writeLines} \quad :: \mathit{Integer} \to [\alpha] \to [[\alpha]] \\
\mathit{writeLines}\ n \ \widehat{=}\ \mathsf{unfoldr}\ (\mathit{split}\ n) \quad \textbf{where}\ \mathit{split}\ n\ [\,] \ \widehat{=}\ \mathsf{Nothing} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{split}\ n\ x \ \widehat{=}\ \mathsf{Just}\ (\mathit{splitAt}\ n\ x)
\end{array}
$$

The function *readLines* is just what is called *concat* in the Haskell standard prelude; we have written it explicitly as a fold here to emphasize the program structure. The function *writeLines* n partitions a list into segments of length n, the last segment possibly being short. (The operator '$\cdot$' denotes function composition, and '$+\!\!+$' is list concatenation.)

The function *reformat* fits our definition of a metamorphism, since it consists of an unfold after a fold. Because $+\!\!+$ is non-strict in its right-hand argument, *reformat* is automatically streaming when lazily evaluated: the first lines of output can be produced before all the input has been consumed. Thus, we need not maintain the whole concatenated list (the result of *readLines*) in memory at once, and we can even reformat infinite lists of lines.

3.2 Radix conversion

Converting fractions from one radix to another is a change of representation. We define functions *radixConvert*, *fromBase* and *toBase* as follows:

$$\begin{aligned}
&\mathit{radixConvert} && :: (\mathit{Integer}, \mathit{Integer}) \to [\mathit{Integer}] \to [\mathit{Integer}] \\
&\mathit{radixConvert}\ (b, b') && \mathrel{\widehat{=}} \mathit{toBase}\ b' \cdot \mathit{fromBase}\ b \\
\\
&\mathit{fromBase} && :: \mathit{Integer} \to [\mathit{Integer}] \to \mathit{Rational} \\
&\mathit{fromBase}\ b && \mathrel{\widehat{=}} \mathsf{foldr}\ \mathit{step}_b\ 0 \\
\\
&\mathit{toBase} && :: \mathit{Integer} \to \mathit{Rational} \to [\mathit{Integer}] \\
&\mathit{toBase}\ b && \mathrel{\widehat{=}} \mathsf{unfoldr}\ \mathit{next}_b
\end{aligned}$$

where

$$\begin{aligned}
&\mathit{step}_b\ n\ x \mathrel{\widehat{=}} (x + n) \div b \\
&\mathit{next}_b\ 0 \mathrel{\widehat{=}} \mathsf{Nothing} \\
&\mathit{next}_b\ x \mathrel{\widehat{=}} \mathsf{Just}\ (\lfloor y \rfloor, y - \lfloor y \rfloor) \quad \textbf{where}\ y \mathrel{\widehat{=}} b \times x
\end{aligned}$$

Thus, *fromBase* b takes a (finite) list of digits and converts it into a fraction; provided the digits are all at least zero and less than b, the resulting fraction will be at least zero and less than one. For example,

$$\mathit{fromBase}\ 10\ [2,5] = \mathit{step}_{10}\ 2\ (\mathit{step}_{10}\ 5\ 0) = {}^{1}\!/_{4}$$

Then *toBase* b takes a fraction between zero and one, and converts it into a (possibly infinite) list of digits in base b. For example,

$$\mathit{toBase}\ 2\ ({}^{1}\!/_{4}) = 0 : \mathsf{unfoldr}\ \mathit{next}_2\ ({}^{1}\!/_{2}) = 0 : 1 : \mathsf{unfoldr}\ \mathit{next}_2\ 0 = [0,1]$$

Composing *fromBase* for one base with *toBase* for another effects a change of base.

At first blush, this looks very similar in structure to the reformatting example of Section 3.1. However, now the fold operator step_b is strict in its right-hand argument. Therefore, *fromBase* b must consume its whole input before it generates any output — so these conversions will not work for infinite fractions, and even for finite fractions the entire input must be read before any output is generated.

Intuitively, one might expect to be able to do better than this. For example, consider converting the decimal fraction $[2,5]$ to the binary fraction $[0,1]$. The initial 2 alone is sufficient to justify the production of the first bit 0 of the output: whatever follows (provided that the input really does consist of decimal digits), the fraction lies between ${}^{2}\!/_{10}$ and ${}^{3}\!/_{10}$, and so its binary representation must start with a zero. We make this intuition precise in Section 4; it involves, among other steps, inverting the structure of the traversal of the input by replacing the foldr with a foldl.

Of course, digit sequences like this are not a good representation for fractions: many useful operations turn out to be uncomputable. In Section 5, we look at a better representation. It still turns out to leave some operations uncomputable (as any non-redundant representation must), but there are fewer of them.

3.3 Heapsort

As a third introductory example, we consider tree-based sorting algorithms. One such sorting algorithm is a variation on Hoare's Quicksort [19]. What makes Quicksort particularly quick is that it is in-place, needing only logarithmic extra space for the control stack; but it is difficult to treat in-place algorithms functionally, so we ignore that aspect. Structurally, Quicksort turns out to be a hylomorphism: it unfolds the input list by repeated partitioning to produce a binary search tree, then folds this tree to yield the output list.

We use the datatype of binary trees from Section 2. We also suppose functions

$$\begin{aligned} &\mathit{partition} :: [\alpha] \to \mathsf{Maybe}\,(\alpha, [\alpha], [\alpha]) \\ &\mathit{join} \quad\;\; :: \mathsf{Maybe}\,(\alpha, [\alpha], [\alpha]) \to [\alpha] \end{aligned}$$

The first partitions a non-empty list into a pivot and the smaller and larger elements (or returns Nothing given an empty list); the second concatenates a pair of lists with a given element in between (or returns the empty list given Nothing); we omit the definitions for brevity. Given these auxiliary functions, we have

$$\mathit{quicksort} \mathrel{\widehat{=}} \mathsf{foldt}\;\mathit{join} \cdot \mathsf{unfoldt}\;\mathit{partition}$$

as a hylomorphism.

One can sort also as a tree metamorphism: the same type of tree is an intermediate data structure, but this time it is a *minheap* rather than a binary search tree: the element stored at each node is no greater than any element in either child of that node. Moreover, this time the tree producer is a *list fold* and the tree consumer is a *list unfold*.

We suppose functions

$$\begin{aligned} &\mathit{insert} \quad :: \alpha \to \mathsf{Tree}\,\alpha \to \mathsf{Tree}\,\alpha \\ &\mathit{splitMin} :: \mathsf{Tree}\,\alpha \to \mathsf{Maybe}\,(\alpha, \mathsf{Tree}\,\alpha) \end{aligned}$$

The first inserts an element into a heap; the second splits a heap into its least element and the remainder (or returns Nothing, given the empty heap). Given these auxilliary functions, we have

$$\mathit{heapsort} \mathrel{\widehat{=}} \mathsf{unfoldr}\;\mathit{splitMin} \cdot \mathsf{foldr}\;\mathit{insert}\;(\mathsf{Node}\;\mathsf{Nothing})$$

as a metamorphism. (Contrast this description of heapsort with the one given by Augusteijn [1] in terms of hylomorphisms, driving the computation by the shape of the intermediate tree rather than the two lists.)

Here, unlike in the reformatting and radix conversion examples, there is no hope for streaming: the second phase cannot possibly make any progress until the entire input is read, because the first element of the sorted output (which is the least element of the list) might be the last element of the input. Sorting is inherently a memory-intensive process, and cannot be performed on infinite lists.

4 Streaming

Of the three examples in Section 3, one automatically permits streaming and one can never do; only one, namely radix conversion, warrants further investigation in this regard. As suggested in Section 3.2, it ought to be possible to produce some of the output before all of the input is consumed. In this section, we see how this can be done, developing some general results along the way.

4.1 The streaming theorem

The second phase of the metamorphism involves producing the output, maintaining some state in the process; that state is initialized to the result of folding the entire input, and evolves as the output is unfolded. Streaming must involve starting to unfold from an earlier state, the result of folding only some initial part of the input. Therefore, it is natural to consider metamorphisms in which the folding phase is an instance of foldl:

$$\mathsf{unfoldr}\ f \cdot \mathsf{foldl}\ g\ c$$

Essentially the problem is a matter of finding some kind of *invariant* of this state 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:*

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

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 (not necessarily strictly) between consuming inputs and producing outputs. We define:

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

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, this input 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 2 (Streaming Theorem [4]). *If the streaming condition holds for f and g, then*

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

on finite lists x.

Proof. The proof is given in [4]. We prove a stronger theorem (Theorem 4) later.

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 main point of introducing streaming in the first place.

As a simple example, consider the functions *unCons* and *snoc*, defined as follows:

$$\begin{aligned} \mathit{unCons}\ [\,] &\mathrel{\widehat{=}} \mathsf{Nothing} \\ \mathit{unCons}\ (a : x) &\mathrel{\widehat{=}} \mathsf{Just}\ (a, x) \\ \mathit{snoc}\ x\ a &\mathrel{\widehat{=}} x +\!\!+ [a] \end{aligned}$$

The streaming condition holds for *unCons* and *snoc*: $\mathit{unCons}\ x = \mathsf{Just}\ (b, x')$ implies $\mathit{unCons}\ (\mathit{snoc}\ x\ a) = \mathsf{Just}\ (b, \mathit{snoc}\ x'\ a)$. Therefore, Theorem 2 applies, and

$$\mathsf{unfoldr}\ \mathit{unCons} \cdot \mathsf{foldl}\ \mathit{snoc}\ [\,] = \mathit{stream}\ \mathit{unCons}\ \mathit{snoc}\ [\,]$$

on finite lists (but not infinite ones!). The left-hand side is a two-stage copying process with an unbounded intermediate buffer, and the right-hand side a one-stage copying queue with a one-place buffer.

4.2 Reversing the order of evaluation

In order to make a streaming version of radix conversion, we need to rewrite *fromBase b* as an instance of foldl rather than of foldr. Fortunately, there is a standard technique for doing this:

$$\mathsf{foldr}\ f\ b = \mathit{applyto}\ b \cdot \mathsf{foldr}\ (\cdot)\ \mathsf{id} \cdot \mathsf{map}\ f$$

where $\mathit{applyto}\ b\ f \mathrel{\widehat{=}} f\ b$. Because composition is associative with unit id, the foldr on the right-hand side can — by the First Duality Theorem [6] — be replaced by a foldl.

Although valid, this is not always very helpful. In particular, it can be quite inefficient — the fold now constructs a long composition of little functions of the

form $f\ a$, and this composition typically cannot be simplified until it is eventually applied to b. However, it is often possible that we can find some *representation* of those little functions that admits composition and application in constant time. Reynolds [34] calls this transformation *defunctionalization*.

Theorem 3. *Given fold arguments $f :: \alpha \to \beta \to \beta$ and $b :: \beta$, suppose there is a type ρ of representations of functions of the form $f\ a$ and their compositions, with the following operations:*

- *a representation function $rep :: \alpha \to \rho$ (so that $rep\ a$ is the representation of $f\ a$);*
- *an abstraction function $abs :: \rho \to \beta \to \beta$, such that $abs\ (rep\ a) = f\ a$;*
- *an analogue $\odot :: \rho \to \rho \to \rho$ of function composition, such that $abs\ (r \odot s) = abs\ r \cdot abs\ s$;*
- *an analogue $ident :: \rho$ of the identity function, such that $abs\ ident = \mathsf{id}$;*
- *an analogue $app_b :: \rho \to \beta$ of application to b, such that $app_b\ r = abs\ r\ b$.*

Then

$$\mathsf{foldr}\ f\ b = app_b \cdot \mathsf{foldl}\ (\odot)\ ident \cdot \mathsf{map}\ rep$$

The foldl *and the map can be fused:*

$$\mathsf{foldr}\ f\ b = app_b \cdot \mathsf{foldl}\ (\circledast)\ ident$$

where $r \circledast a \mathrel{\widehat{=}} r \odot rep\ a$.

(Note that the abstraction function *abs* is used above only for stating the correctness conditions; it is not applied anywhere.)

For example, let us return to radix conversion, as introduced in Section 3.2. The 'little functions' here are of the form $step_b\ n$, or equivalently, $(\div b) \cdot (+n)$. This class of functions is closed under composition:

$$\begin{array}{ll} & (step_c\ n \cdot step_b\ m)\ x \\ = & \quad \{\text{composition}\} \\ & step_c\ n\ (step_b\ m\ x) \\ = & \quad \{step\} \\ & ((x + m) \div b + n) \div c \\ = & \quad \{\text{arithmetic}\} \\ & (x + m + b \times n) \div (b \times c) \\ = & \quad \{\text{composition}\} \\ & ((\div(b \times c)) \cdot (+m + b \times n))\ x \end{array}$$

We therefore defunctionalize $step_b\ n$ to the pair (n, b), and define:

$$\begin{array}{ll} rep_b\ n & \mathrel{\widehat{=}} (n, b) \\ abs\ (n, b)\ x & \mathrel{\widehat{=}} (x + n) \div b \\ (n, c) \odot (m, b) & \mathrel{\widehat{=}} (m + b \times n, b \times c) \\ (n, c) \circledast_b m & \mathrel{\widehat{=}} (n, c) \odot rep_b\ m \ \mathrel{\widehat{=}}\ (m + b \times n, b \times c) \\ ident & \mathrel{\widehat{=}} (0, 1) \\ app\ (n, b) & \mathrel{\widehat{=}} abs\ (n, b)\ 0 \ \mathrel{\widehat{=}}\ n \div b \end{array}$$

Theorem 3 then tells us that

$$fromBase\ b = app \cdot \mathsf{foldl}\ (\circledast_b)\ ident$$

4.3 Checking the streaming condition

We cannot quite apply Theorem 2 yet, because the composition of *toBase* b' and the revised *fromBase* b has the abstraction function *app* between the unfold and the fold. Fortunately, that *app* fuses with the unfold. For brevity below, we define

$$\begin{aligned} &\mathit{mapl}\ f\ \mathsf{Nothing} &&\mathrel{\widehat{=}}\ \mathsf{Nothing} \\ &\mathit{mapl}\ f\ (\mathsf{Just}\ (a,b)) &&\mathrel{\widehat{=}}\ \mathsf{Just}\ (a, f\ b) \end{aligned}$$

(that is, *mapl* is the map operation of the base functor for the list datatype); then

$$\begin{aligned} &\mathsf{unfoldr}\ \mathit{next}_c \cdot \mathit{app} = \mathsf{unfoldr}\ \mathit{nextapp}_c \\ \Leftarrow\quad &\{\text{unfold fusion}\} \\ &\mathit{next}_c \cdot \mathit{app} = \mathit{mapl}\ \mathit{app} \cdot \mathit{nextapp}_c \end{aligned}$$

and

$$\begin{aligned} &\mathit{next}_c\ (\mathit{app}\ (n,r)) \\ =\quad &\{\mathit{app},\ \mathit{next}_c;\ \text{let}\ u \mathrel{\widehat{=}} \lfloor n \times c \div r \rfloor\} \\ &\mathbf{if}\ n{=}{=}0\ \mathbf{then}\ \mathsf{Nothing}\ \mathbf{else}\ \mathsf{Just}\ (u, n{\times}c{\div}r - u) \\ =\quad &\{\mathit{app};\ \text{there is some leeway here (see below)}\} \\ &\mathbf{if}\ n{=}{=}0\ \mathbf{then}\ \mathsf{Nothing}\ \mathbf{else}\ \mathsf{Just}\ (u, \mathit{app}\ (n - u{\times}r{\div}c, r{\div}c)) \\ =\quad &\{\mathit{mapl}\} \\ &\mathit{mapl}\ \mathit{app}\ (\mathbf{if}\ n{=}{=}0\ \mathbf{then}\ \mathsf{Nothing}\ \mathbf{else}\ \mathsf{Just}\ (u, (n - u{\times}r{\div}c, r{\div}c))) \end{aligned}$$

Therefore we try defining

$$\begin{aligned} \mathit{nextapp}_c\ (n,r) \mathrel{\widehat{=}}\ &\mathbf{if}\ n{=}{=}0\ \mathbf{then}\ \mathsf{Nothing}\ \mathbf{else}\ \mathsf{Just}\ (u, (n - u{\times}r{\div}c, r{\div}c)) \\ &\mathbf{where}\ u \mathrel{\widehat{=}} \lfloor n \times c \div r \rfloor \end{aligned}$$

Note that there was some leeway here: we had to partition the rational $n{\times}c{\div}r{-}u$ into a numerator and denominator, and we chose $(n - u{\times}r{\div}c, r{\div}c)$ out of the many ways of doing this. One might perhaps have expected $(n{\times}c - u{\times}r, r)$ instead; however, this leads to a dead-end, as we show later. Note that our choice involves generalizing from integer to rational components.

Having now massaged our radix conversion program into the correct format:

$$\mathit{radixConvert}\ (b,c) = \mathsf{unfoldr}\ \mathit{nextapp}_c \cdot \mathsf{foldl}\ (\circledast_b)\ \mathit{ident}$$

we may consider whether the streaming condition holds for $\mathit{nextapp}_c$ and $\circledast_b$; that is, whether

$$\begin{aligned} &\mathit{nextapp}_c\ (n,r) = \mathsf{Just}\ (u, (n', r')) \\ \Rightarrow\quad &\\ &\mathit{nextapp}_c\ ((n,r) \circledast_b m) = \mathsf{Just}\ (u, (n', r') \circledast_b m) \end{aligned}$$

An element u is produced from a state (n,r) iff $n \neq 0$, in which case $u = \lfloor n \times c \div r \rfloor$. The modified state $(n,r) \circledast_b m$ evaluates to $(m + b \times n, b \times r)$. Since $n, b > 0$ and $m \geq 0$, this necessarily yields an element; this element v equals

$\lfloor (m + b \times n) \times c \div (b \times r) \rfloor$. We have to check that u and v are equal. Sadly, in general they are not: since $0 \leq m < b$, it follows that v lies between u and $\lfloor (n+1) \times c \div r \rfloor$, but these two bounds need not meet.

Intuitively, this can happen when the state has not completely determined the next output, and further inputs are needed in order to make a commitment to that output. For example, consider having consumed the first digit 6 while converting the sequence $[6, 7]$ in decimal (representing 0.67_{10}) to ternary. The fraction 0.6_{10} is about 0.1210_3; nevertheless, it is not safe to commit to producing the digit 1, because the true result is greater than 0.2_3, and there is not enough information to decide whether to output a 1 or a 2 until the 7 has been consumed as well.

This is a common situation with streaming algorithms: the producer function (*nextapp* above) 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. That is the subject of the next section.

4.4 Flushing streams

The typical approach is to introduce a 'restriction'

$$snextapp = \mathsf{guard}\ safe\ nextapp$$

of *nextapp* for some predicate *safe*, where

$$\mathsf{guard}\ p\ f\ x \mathrel{\widehat{=}} \mathbf{if}\ p\ x\ \mathbf{then}\ f\ x\ \mathbf{else}\ \mathsf{Nothing}$$

and to use *snextapp* as the producer for the streaming process. In the case of radix conversion, the predicate $safe_c$ (dependent on the output base c) could be defined

$$safe_c\ (n, r) \mathrel{\widehat{=}} (\lfloor n \times c \div r \rfloor == \lfloor (n+1) \times c \div r \rfloor)$$

That is, the state (n, r) is safe for the output base c if these lower and upper bounds on the next digit meet; with this proviso, the streaming condition holds, as we checked above. (In fact, we need to check not only that the same elements are produced from the unmodified and the modified state, but also that the two residual states are related in the same way as the two original states. With the definition of $nextapp_c$ that we chose above, this second condition does hold; with the more obvious definition involving $(n \times c - u \times r, r)$ that we rejected, it does not.)

However, with this restricted producer the streaming process no longer has the same behaviour on finite lists as does the plain metamorphism: when the input is exhausted, the more cautious *snextapp* may have left some outputs

still to be produced that the more aggressive *nextapp* would have emitted. The streaming process should therefore switch into a final 'flushing' phase when all the input has been consumed.

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

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

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 the state 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*.

The relationship of metamorphisms to flushing streams is a little more complicated than that to ordinary streams. One way of expressing the relationship is via a generalization of $\mathsf{unfoldr}$, whose final action is to generate a whole tail of the resulting list rather than the empty list. This is an instance of *primitive corecursion* (called an *apomorphism* by Vene and Uustalu [37]), which is the categorical dual of *primitive recursion* (called a *paramorphism* by Meertens [29]).

$$
\begin{array}{l}
\mathsf{apol} \quad\ :: (\beta \to \mathsf{Maybe}\,(\alpha,\beta)) \to (\beta \to [\alpha]) \to \beta \to [\alpha] \\
\mathsf{apol}\ f\ h\ b \mathrel{\widehat{=}} \mathbf{case}\ f\ b\ \mathbf{of} \\
\qquad\qquad\qquad\quad \mathsf{Just}\,(a, b') \to a : \mathsf{apol}\ f\ h\ b' \\
\qquad\qquad\qquad\quad \mathsf{Nothing} \quad\ \to h\ b
\end{array}
$$

Informally, $\mathsf{apol} f\, h\, b = \mathsf{unfoldr} f\, b +\!\!+ h\, b'$, where b' is the final state of the unfold (if there is one — and if there is not, the value of $h\ b'$ is irrelevant), and $\mathsf{unfoldr}\, f = \mathsf{apol}\, f\, (\mathsf{const}\,[\,])$. On finite inputs, provided that the streaming condition holds, a flushing stream process yields the same result as the ordinary streaming process, but with the results of flushing the final state (if any) appended.

Theorem 4 (Flushing Stream Theorem). *If the streaming condition holds for f and g, then*

$$\mathit{fstream}\ f\ g\ h\ c\ x = \mathsf{apol}\, f\, h\, (\mathsf{foldl}\ g\ c\ x)$$

on finite lists x.

The proof uses the following lemma [4], which lifts the streaming condition from single inputs to finite lists of inputs.

Lemma 5. *If the streaming condition holds for f and g, then*

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

for all b, c, c' and finite lists x.

It also uses the *approximation lemma* [5, 15].

Lemma 6 (Approximation Lemma). *For finite, infinite or partial lists x and y,*

$$x = y \equiv \forall n.\ \mathit{approx}\ n\ x = \mathit{approx}\ n\ y$$

where

$$\begin{array}{ll} \mathit{approx} & :: \mathit{Int} \to [\alpha] \to [\alpha] \\ \mathit{approx}\ (n+1)\ [\,] & = [\,] \\ \mathit{approx}\ (n+1)\ (a : x) & = a : \mathit{approx}\ n\ x \end{array}$$

(Note that $\mathit{approx}\ 0\ x = \bot$ for any x, by case exhaustion.)

Proof (of Theorem 4). By Lemma 6, it suffices to show, for fixed f, g, h and for all n and finite x, that

$$\forall c.\ \mathit{approx}\ n\ (\mathit{fstream}\ f\ g\ h\ c\ x) = \mathit{approx}\ n\ (\mathsf{apol}\ f\ h\ (\mathsf{foldl}\ g\ c\ x))$$

under the assumption that the streaming condition holds for f and g. We use a 'double induction' simultaneously over n and the length $\#x$ of x. The inductive hypothesis is that

$$\forall c.\ \mathit{approx}\ m\ (\mathit{fstream}\ f\ g\ h\ c\ y) = \mathit{approx}\ m\ (\mathsf{apol}\ f\ h\ (\mathsf{foldl}\ g\ c\ y))$$

for any m, y such that $m < n \wedge \#y \leq \#x$ or $m \leq n \wedge \#y < \#x$. We then proceed by case analysis to complete the inductive step.

Case $f\ c = \mathsf{Just}\ (b, d)$**.** In this case, we make a subsidiary case analysis on n.

Subcase $n = 0$**.** Then the result holds trivially.

Subcase $n = n' + 1$**.** Then we have:

$$\begin{array}{ll} & \mathit{approx}\ (n'+1)\ (\mathsf{apol}\ f\ h\ (\mathsf{foldl}\ g\ c\ x)) \\ = & \quad \{\text{Lemma 5: } f\ (\mathsf{foldl}\ g\ c\ x) = \mathsf{Just}\ (b, \mathsf{foldl}\ g\ d\ x)\} \\ & \mathit{approx}\ (n'+1)\ (b : \mathsf{apol}\ f\ h\ (\mathsf{foldl}\ g\ d\ x)) \\ = & \quad \{\mathit{approx}\} \\ & b : \mathit{approx}\ n'\ (\mathsf{apol}\ f\ h\ (\mathsf{foldl}\ g\ d\ x)) \\ = & \quad \{\text{induction: } n' < n\} \\ & b : \mathit{approx}\ n'\ (\mathit{fstream}\ f\ g\ h\ d\ x) \\ = & \quad \{\mathit{approx}\} \\ & \mathit{approx}\ (n'+1)\ (b : \mathit{fstream}\ f\ g\ h\ d\ x) \\ = & \quad \{\mathit{fstream}\text{; case assumption}\} \\ & \mathit{approx}\ (n'+1)\ (\mathit{fstream}\ f\ g\ h\ c\ x) \end{array}$$

Case $f\ c = \mathsf{Nothing}$**.** In this case, we make a subsidiary case analysis on x.

Subcase $x = a : x'$**.** Then

$$\begin{array}{ll} & \mathsf{apol}\ f\ h\ (\mathsf{foldl}\ g\ c\ (a : x')) \\ = & \quad \{\mathsf{foldl}\} \\ & \mathsf{apol}\ f\ h\ (\mathsf{foldl}\ g\ (g\ c\ a)\ x')) \\ = & \quad \{\text{induction: } \#x' < \#x\} \\ & \mathit{fstream}\ f\ g\ h\ (g\ c\ a)\ x' \\ = & \quad \{\mathit{fstream}\text{; case assumption}\} \\ & \mathit{fstream}\ f\ g\ h\ c\ (a : x') \end{array}$$

Subcase $x = [\,]$**.** Then

$$\begin{array}{ll} & \mathsf{apol}\ f\ h\ (\mathsf{foldl}\ g\ c\ [\,]) \\ = & \quad \{\mathsf{foldl}\} \\ & \mathsf{apol}\ f\ h\ c \\ = & \quad \{\text{case assumption}\} \\ & h\ c \\ = & \quad \{\mathit{fstream};\ \text{case assumption}\} \\ & \mathit{fstream}\ f\ g\ h\ c\ [\,] \end{array}$$

4.5 Invoking the Flushing Stream Theorem

Theorem 4 gives conditions under which an apomorphism applied to the result of a foldl may be streamed. This seems of limited use, since such scenarios are not commonly found. However, they can be constructed from more common scenarios in which the apomorphism is replaced with a simpler unfold. One way is to introduce the trivial apomorphism, whose flusher always returns the empty list. A more interesting, and the most typical, way is via the observation that

$$\mathsf{apol}\ (\mathsf{guard}\ p\ f)\ (\mathsf{unfoldr}\ f) = \mathsf{unfoldr}\ f$$

for any predicate p. Informally, the work of an unfold can be partitioned into 'cautious' production, using the more restricted producer $\mathsf{guard}\ p\ f$, followed by more 'aggressive' production using simply f when the more cautious producer blocks.

4.6 Radix conversion as a flushing stream

Returning for a final time to radix conversion, we define

$$\mathit{snextapp}_c\ (n, r) \mathrel{\widehat{=}} \mathsf{guard}\ \mathit{safe}_c\ \mathit{nextapp}_c$$

We verified in Sections 4.3 and 4.4 that the streaming condition holds for $\mathit{snextapp}_c$ and $\circledast_b$. Theorem 4 then tells us that we can convert from base b to base c using

$$\mathit{radixConvert}\ (b, c) = \mathit{fstream}\ \mathit{snextapp}_c\ (\circledast_b)\ (\mathsf{unfoldr}\ \mathit{nextapp}_c)\ (0, 1)$$

This program works for finite or infinite inputs, and is always productive. (It does, however, always produce an infinite result, even when a finite result would be correct. For example, it will correctly convert $\frac{1}{3}$ from base 10 to base 2, but in converting from base 10 to base 3 it will produce an infinite tail of zeroes. One cannot really hope to do better, as returning a finite output depending on the entire infinite input is uncomputable.)

5 Continued fractions

Continued fractions are finite or infinite constructions of the form

$$b_0 + \cfrac{a_0}{b_1 + \cfrac{a_1}{b_2 + \cfrac{a_2}{b_3 + \cdots}}}$$

in which all the coefficients are integers. They provide an elegant representation of numbers, both rational and irrational. They have therefore been proposed by various authors [2, 17, 24, 38, 26, 27] as a good format in which to carry out *exact real arithmetic*. Some of the algorithms for simple arithmetic operations on continued fractions can be seen as metamorphisms, and as we shall show here, they can typically be streamed.

We consider algorithms on *regular* continued fractions: ones in which all the a_i coefficients are 1, and all the b_i coefficients (except perhaps b_0) are at least 1. We denote regular continued fractions more concisely in the form $\langle b_0, b_1, b_2, \ldots \rangle$. For example, the continued fraction for π starts $\langle 3, 7, 15, 1, 292, \ldots \rangle$. Finite continued fractions correspond to rationals; infinite continued fractions represent irrational numbers.

5.1 Converting continued fractions

We consider first conversions between rationals and finite regular continued fractions. To complete the isomorphism between these two sets, we need to augment the rationals with $1/0 = \infty$, corresponding to the empty continued fraction. We therefore introduce a type *ExtRat* of rationals extended with ∞.

Conversion from rationals to continued fractions is straightforward. Infinity, by definition, is represented by the empty fraction. A finite rational a/b has a first term given by $a \mathbin{\underline{div}} b$, the integer obtained by rounding the fraction down; this leaves a remainder of $(a \mathbin{\underline{mod}} b)/b$, whose reciprocal is the rational from which to generate the remainder of the continued fraction. Note that as a consequence of rounding the fraction down to get the first term, the remainder is between zero and one, and its reciprocal is at least one; therefore the next term (and by induction, all subsequent terms) will be at least one, yielding a regular continued fraction as claimed.

```
type CF = [Integer]

toCF  :: ExtRat → CF
toCF  ≙ unfoldr get
      where get x ≙ if    x==∞
                  then Nothing
                  else Just (⌊x⌋, 1/(x−⌊x⌋))
```

Converting in the opposite direction is more difficult: of course, not all continued fractions correspond to rationals. However, finite ones do, and for these

we can compute the rational using a fold — it suffices to fold with the inverse of *get* (or at least, what would have been the inverse of *get*, if foldr had been defined to take an argument of type $\mathsf{Maybe}\,(\alpha,\beta) \to \beta$, dualizing unfoldr).

$$\begin{array}{l} \mathit{fromCF} :: \mathit{CF} \to \mathit{ExtRat} \\ \mathit{fromCF} \mathrel{\widehat{=}} \mathsf{foldr}\ \mathit{put}\ \infty \quad \textbf{where}\ \mathit{put}\ n\ y \mathrel{\widehat{=}} n + {}^{1}\!/_{y} \end{array}$$

Thus, $\mathit{fromCF} \cdot \mathit{toCF}$ is the identity on extended rationals, and $\mathit{toCF} \cdot \mathit{fromCF}$ is the identity on finite continued fractions. On infinite continued fractions, *fromCF* yields no result: *put* is strict, so the whole list of coefficients is required. One could compute an infinite sequence of rational approximations to the irrational value represented by an infinite continued fraction, by converting to a rational each of the convergents. But this is awkward, because the fold starts at the right, and successive approximations will have no common subexpressions — it does not constitute a scan. It would be preferable if we could write *fromCF* as an instance of foldl; then the sequence of approximations would be given as the corresponding scanl.

Fortunately, Theorem 3 comes to the rescue again. This requires defunctionalizations of functions of the form $\mathit{put}\ n$ and their compositions. For proper rationals, we reason:

$$\begin{array}{ll} & \mathit{put}\ n\ (\mathit{put}\ m\ {}^{a}\!/_{b}) \\ = & \quad \{\mathit{put}\} \\ & \mathit{put}\ n\ (m + {}^{b}\!/_{a}) \\ = & \quad \{\text{arithmetic}\} \\ & \mathit{put}\ n\ ({}^{m \times a + b}\!/_{a}) \\ = & \quad \{\mathit{put}\} \\ & n + {}^{a}\!/_{m \times a + b} \\ = & \quad \{\text{arithmetic}\} \\ & (n \times (m \times a + b) + a)/(m \times a + b) \\ = & \quad \{\text{collecting terms; dividing through by } b\} \\ & ((n \times m + 1) \times {}^{a}\!/_{b} + n)/(m \times {}^{a}\!/_{b} + 1) \end{array}$$

This is a ratio of integer-coefficient linear functions of ${}^{a}\!/_{b}$, sometimes known as a *rational function* or *linear fractional transformation* of ${}^{a}\!/_{b}$. The general form of such a function takes x to $(q\,x + r)/(s\,x + t)$ (denoting multiplication by juxtaposition for brevity), and can be represented by the four integers q, r, s, t.

For the improper rational ∞, we reason:

$$\begin{array}{ll} & \mathit{put}\ n\ (\mathit{put}\ m\ \infty) \\ = & \quad \{\mathit{put}\} \\ & \mathit{put}\ n\ (m + {}^{1}\!/_{\infty}) \\ = & \quad \{{}^{1}\!/_{\infty} = 0\} \\ & \mathit{put}\ n\ m \\ = & \quad \{\mathit{put}\} \\ & n + {}^{1}\!/_{m} \\ = & \quad \{\text{arithmetic}\} \\ & (n \times m + 1)/m \end{array}$$

which agrees with the result for proper rationals, provided we take the reasonable interpretation that $(q \times {}^{a}/_{b} + r)/(s \times {}^{a}/_{b} + t) = {}^{q \times a + r \times b}/_{s \times a + t \times b}$ when $b = 0$.

Following Theorem 3, then, we choose four-tuples of integers as our representation; for reasons that will become clear, we write these four-tuples in the form $\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)$. The abstraction function *abs* applies the rational function:

$$abs \left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right) x \mathrel{\widehat{=}} {}^{q\,x+r}/_{s\,x+t}$$

and the representation function *rep* injects the integer n into the representation of *put* n:

$$rep\ n \mathrel{\widehat{=}} \left(\begin{smallmatrix} n & 1 \\ 1 & 0 \end{smallmatrix}\right)$$

The identity function is represented by *ident*:

$$ident \mathrel{\widehat{=}} \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$$

We verify that rational functions are indeed closed under composition, by constructing the representation of function composition:

$$\begin{array}{ll} & abs\ (\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right) \odot \left(\begin{smallmatrix} q' & r' \\ s' & t' \end{smallmatrix}\right))\ x \\ = & \quad \{\text{requirement}\} \\ & abs \left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)\ (abs \left(\begin{smallmatrix} q' & r' \\ s' & t' \end{smallmatrix}\right)\ x) \\ = & \quad \{abs\} \\ & abs \left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)\ ((q'\,x + r')/(s'\,x + t')) \\ = & \quad \{abs \text{ again}\} \\ & (q\,(q'\,x{+}r') + r\,(s'\,x{+}t'))/(s\,(q'\,x{+}r') + t\,(s'\,x{+}t')) \\ = & \quad \{\text{collecting terms}\} \\ & ((q\,q'{+}r\,s')\,x + (q\,r'{+}r\,t'))/((s\,q'{+}t\,s')\,x + (s\,r'{+}t\,t')) \\ = & \quad \{abs\} \\ & abs \left(\begin{smallmatrix} q\,q'{+}r\,s' & q\,r'{+}r\,t' \\ s\,q'{+}t\,s' & s\,r'{+}t\,t' \end{smallmatrix}\right)\ x \end{array}$$

We therefore define

$$\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right) \odot \left(\begin{smallmatrix} q' & r' \\ s' & t' \end{smallmatrix}\right) \mathrel{\widehat{=}} \left(\begin{smallmatrix} q\,q'{+}r\,s' & q\,r'{+}r\,t' \\ s\,q'{+}t\,s' & s\,r'{+}t\,t' \end{smallmatrix}\right)$$

Finally, we define an extraction function

$$\begin{array}{ll} app \left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right) \mathrel{\widehat{=}} & abs \left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right) \infty \\ = & {}^{q}/_{s} \end{array}$$

(Notice that $\odot$ turns out to be matrix multiplication, and *ident* the unit matrix, which explains the choice of notation. These matrices are sometimes called *homographies*, and the rational functions they represent *homographic functions* or *Möbius transformations*. They can be generalized from continued fractions to many other interesting exact representations of real numbers [32], including redundant ones. In fact, the same framework also encompasses radix conversions, as explored in Section 3.2.)

By Theorem 3 we then have

$$fromCF = app \cdot \mathsf{foldl}\ (\circledast)\ ident \quad \textbf{where}\ \left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right) \circledast n \mathrel{\widehat{=}} \left(\begin{smallmatrix} n\,q{+}r & q \\ n\,s{+}t & s \end{smallmatrix}\right)$$

Of course, this still will not work for infinite continued fractions; however, we can now define

$$\begin{aligned}&\mathit{fromCFi} :: \mathit{CF} \to [\mathit{ExtRat}]\\ &\mathit{fromCFi} \mathrel{\widehat{=}} \mathsf{map}\ \mathit{app} \cdot \mathsf{scanl}\ (\circledast)\ \mathit{ident}\end{aligned}$$

yielding the (infinite) sequence of finite convergents of an (infinite) continued fraction.

5.2 Rational unary functions of continued fractions

In Section 5.1, we derived the program

$$\mathit{fromCF} = \mathit{app} \cdot \mathsf{foldl}\ (\circledast)\ \left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)$$

for converting a finite continued fraction to an extended rational. In fact, we can compute an arbitrary rational function of a continued fraction, by starting this process with an arbitrary homography in place of the identity $\left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)$. This is because composition $\odot$ fuses with the fold:

$$\begin{aligned}&\mathit{abs}\ h\ (\mathit{fromCF}\ \mathit{ns})\\ =\quad &\{\mathit{fromCF}\}\\ &\mathit{abs}\ h\ (\mathit{app}\ (\mathsf{foldl}\ (\circledast)\ \mathit{ident}\ \mathit{ns}))\\ =\quad &\{\text{specification of } \mathit{app}\}\\ &\mathit{abs}\ h\ (\mathit{abs}\ (\mathsf{foldl}\ (\circledast)\ \mathit{ident}\ \mathit{ns})\ \infty)\\ =\quad &\{\text{requirement on } \mathit{abs} \text{ and } \odot\}\\ &\mathit{abs}\ (h \odot \mathsf{foldl}\ (\circledast)\ \mathit{ident}\ \mathit{ns})\ \infty\\ =\quad &\{\text{fold fusion: } \odot \text{ is associative, and } \mathit{ident} \text{ its unit}\}\\ &\mathit{abs}\ (\mathsf{foldl}\ (\circledast)\ h\ \mathit{ns})\ \infty\\ =\quad &\{\text{specification of } \mathit{app} \text{ again}\}\\ &\mathit{app}\ (\mathsf{foldl}\ (\circledast)\ h\ \mathit{ns})\end{aligned}$$

For example, suppose we want to compute the rational $^2/_{x-3}$, where x is the rational represented by a particular (finite) continued fraction ns. We could convert ns to the rational x, then perform the appropriate rational arithmetic. Alternatively, we could convert ns to a rational as above, starting with the homography $\left(\begin{smallmatrix}0 & 2\\ 1 & -3\end{smallmatrix}\right)$ instead of $\left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)$, and get the answer directly. If we want the result as a continued fraction again rather than a rational, we simply post-apply toCF.

Of course, this will not work to compute rational functions of *infinite* continued fractions, as the folding will never yield a result. Fortunately, it is possible to applying streaming, so that terms of the output are produced before the whole input is consumed. This is the focus of the remainder of this section. The derivation follows essentially the same steps as were involved in radix conversion.

The streaming process maintains a state in the form of a homography, which represents the mapping from what is yet to be consumed to what is yet to be produced. The production steps of the streaming process choose a term to output, and compute a reduced homography for the remainder of the computation.

Given a current homography $\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)$, and a chosen term n, the reduced homography $\left(\begin{smallmatrix} q' & r' \\ s' & t' \end{smallmatrix}\right)$ is determined as follows:

$$\begin{array}{ll} & (q\,x+r)/(s\,x+t) = n + 1/((q'\,x+r')/(s'\,x+t')) \\ \equiv & \quad \{\text{reciprocal}\} \\ & (q\,x+r)/(s\,x+t) = n + (s'\,x+t')/(q'\,x+r') \\ \equiv & \quad \{\text{rearrange}\} \\ & (s'\,x+t')/(q'\,x+r') = (q\,x+r)/(s\,x+t) - n \\ \equiv & \quad \{\text{incorporate } n \text{ into fraction}\} \\ & (s'\,x+t')/(q'\,x+r') = (q\,x+r-n\,(s\,x+t))/(s\,x+t) \\ \equiv & \quad \{\text{collect } x \text{ and non-}x \text{ terms}\} \\ & (s'\,x+t')/(q'\,x+r') = ((q-n\,s)\,x+r-n\,t)/(s\,x+t) \\ \Leftarrow & \quad \{\text{equating terms}\} \\ & q' = s, r' = t, s' = q - n\,s, t' = r - n\,t \end{array}$$

That is,

$$\left(\begin{smallmatrix} q' & r' \\ s' & t' \end{smallmatrix}\right) = \left(\begin{smallmatrix} 0 & 1 \\ 1 & -n \end{smallmatrix}\right) \odot \left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)$$

We therefore define

$$\begin{array}{rl} \mathit{emit}\ \left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)\ n \mathrel{\widehat{=}} & \left(\begin{smallmatrix} 0 & 1 \\ 1 & -n \end{smallmatrix}\right) \odot \left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right) \\ = & \left(\begin{smallmatrix} s & t \\ q-n\,s & r-n\,t \end{smallmatrix}\right) \end{array}$$

Making it a metamorphism. In most of what follows, we assume that we have a *completely regular* continued fraction, namely one in which every coefficient including the first is at least one. This implies that the value represented by the continued fraction is between one and infinity. We see at the end of the section what to do about the first coefficient, in case it is less than one.

Given the representation of a rational function in the form of a homography h, we introduce the function *rfc* ('*r*ational *f*unction of a *c*ompletely regular continued fraction') to apply it as follows:

$$\mathit{rfc}\ h \mathrel{\widehat{=}} \mathit{toCF} \cdot \mathit{app} \cdot \mathsf{foldl}\ (\circledast)\ h$$

This is almost a metamorphism: *toCF* is indeed an unfold, but we must get rid of the projection function *app* in the middle. Fortunately, it fuses with the unfold:

$$\mathsf{unfoldr}\ \mathit{get} \cdot \mathit{app} = \mathsf{unfoldr}\ \mathit{geth}$$

where *geth* (for '*get* on homographies') is defined by

$$\begin{array}{rl} \mathit{geth}\ \left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right) \mathrel{\widehat{=}} & \textbf{if}\ s{=}{=}0\ \textbf{then}\ \mathsf{Nothing}\ \textbf{else}\ \mathsf{Just}\ (n, \mathit{emit}\ \left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)\ n) \\ & \quad \textbf{where}\ n \mathrel{\widehat{=}} q\ \underline{\mathit{div}}\ s \end{array}$$

as can easily be verified.

This yields a metamorphism:

$$\mathit{rfc}\ h = \mathsf{unfoldr}\ \mathit{geth} \cdot \mathsf{foldl}\ (\circledast)\ h$$

Checking the streaming condition. Now we must check that the streaming condition holds for *geth* and $\circledast$. We require that when

$$geth\ h = \mathsf{Just}\ (n, h')$$

then, for any subsequent term m (which we can assume to be at least 1, this being a completely regular continued fraction),

$$geth\ (h \circledast m) = \mathsf{Just}\ (n, h' \circledast m)$$

Unpacking this, when $h = \left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)$ and $h' = \left(\begin{smallmatrix} q' & r' \\ s' & t' \end{smallmatrix}\right)$, we have $s \neq 0$, $n = q \mathbin{\underline{div}} s$, $q' = s$, and $s' = q \mathbin{\underline{mod}} s$; moreover, $\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right) \circledast m = \left(\begin{smallmatrix} m\,q+r & q \\ m\,s+t & s \end{smallmatrix}\right)$. We require among other things that $m\,s + t \neq 0$ and $(m\,q + r) \mathbin{\underline{div}} (m\,s + t) = q \mathbin{\underline{div}} s$. Sadly, this does not hold; for example, if $m = 1$ and s, t are positive,

$$^{q+r}/_{s+t} < 1 + {}^{q}/_{s} \;\equiv\; s\,(q + r) < (q + s)\,(q + t) \;\equiv\; r\,s < q\,t + s\,t + s^2$$

which fails if r is sufficiently large.

Cautious progress. As with the radix conversion algorithm in Section 4.3, the function that produces the next term of the output must be more cautious when it is interleaved with consumption steps that it may be after all the input has been consumed. The above discussion suggests that we should commit to an output only when it is safe from being invalidated by a later input; in symbols, only when $(m\,q + r) \mathbin{\underline{div}} (m\,s + t) = q \mathbin{\underline{div}} s$ for any $m \geq 1$. This follows if s and t are non-zero and have the same sign, and if $(q + r) \mathbin{\underline{div}} (s + t) = q \mathbin{\underline{div}} s$, as a little calculation will verify.

(Another way of looking at this is to observe that the value represented by a completely regular continued fraction ranges between 1 and ∞, so the result of transforming it under a homography $\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)$ ranges between

$$abs\ \left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)\ 1 = {}^{q+r}/_{s+t}$$

and

$$abs\ \left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)\ \infty = {}^{q}/_{s}$$

if $s \neq 0$. If the two denominators have the same sign, the result ranges between these two; if they have different signs, it ranges outside them. Therefore, the first coefficient of the output is determined if the denominators have the same sign (which follows if s and t are non-zero and of the same sign) and the two fractions have the same integer parts.)

We therefore define *gets* (for 'safe *get*') by

$$\begin{aligned} gets\ \left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right) \mathbin{\widehat{=}} \mathbf{let}\ & n \mathbin{\widehat{=}} q \mathbin{\underline{div}} s\ \mathbf{in} \\ & \mathbf{if} \quad s\,t > 0 \wedge (q + r) \mathbin{\underline{div}} (s + t) = n \\ & \mathbf{then}\ \mathsf{Just}\ (n, emit\ \left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)\ n) \\ & \mathbf{else}\ \ \mathsf{Nothing} \end{aligned}$$

Note that whenever *gets* produces a value, *geth* produces the same value; but sometimes *gets* produces nothing when *geth* produces something. The streaming condition *does* hold for *gets* and $\circledast$, as the reader may now verify.

Flushing streams. It is not the case that $\mathsf{unfoldr}\ get \cdot app = \mathsf{unfoldr}\ gets$, of course, because the latter is too cautious. However, it does follow that

$$\mathsf{unfoldr}\ get \cdot app = apol\ gets\ (\mathsf{unfoldr}\ geth)$$

This cautiously produces elements while it it safe to do so, then throws caution to the winds and produces elements anyway when it ceases to be safe. Moreover, Theorem 4 applies to the cautious part, and so

$$\begin{aligned} rfc\ h &= \mathsf{unfoldr}\ get \cdot app \cdot \mathsf{foldl}\ (\circledast)\ h \\ &= fstream\ gets\ (\circledast)\ (\mathsf{unfoldr}\ geth)\ h \end{aligned}$$

This streaming algorithm can compute a rational function of a finite or infinite completely regular continued fraction, yielding a finite or infinite regular continued fraction as a result.

Handling the first term. A regular but not completely regular continued fraction may have a first term of 1 or less, invalidating the reasoning above. However, this is easy to handle, simply by consuming the first term immediately. We introduce a wrapper function rf:

$$\begin{aligned} rf\ h\ [\,] &\mathrel{\widehat{=}} rfc\ h\ [\,] \\ rf\ h\ (n : x) &\mathrel{\widehat{=}} rfc\ (h \circledast n)\ x \end{aligned}$$

This streaming algorithm can compute any rational function of a finite or infinite regular continued fraction, completely regular or not.

5.3 Rational binary functions of continued fractions

The streaming process described in Section 5.2 allows us to compute a unary rational function $(a\,x+b)/(c\,x+d)$ of a single continued fraction x. The technique can be adapted to allow a binary rational function $(a\,x\,y+b\,x+c\,y+d)/(e\,x\,y+f\,x+g\,y+h)$ of continued fractions x and y. This does not fit into our framework of metamorphisms and streaming algorithms, because it combines two arguments into one result; nevertheless, much of the same reasoning can be applied. We intend to elaborate on this in a companion paper.

6 Two other applications

In this section, we briefly outline two other applications of streaming; we have described both in more detail elsewhere, and we refer the reader to those sources for the details.

6.1 Digits of π

In [14] we present an *unbounded spigot algorithm* for computing the digits of π. This work was inspired by Rabinowitz and Wagon [33], who coined the term *spigot algorithm* for an algorithm that yields output elements incrementally and does not reuse them after they have been output — so the digits drip out one by one, as if from a leaky tap. (In contrast, most algorithms for computing approximations to π, including the best currently known, work inscrutably until they deliver a complete response at the end of the computation.) Although incremental, Rabinowitz and Wagon's algorithm is *bounded*, since one needs to decide at the outset how many digits are to be computed, whereas our algorithm yields digits indefinitely. (This is nothing to do with evaluation order: Rabinowitz and Wagon's algorithm is just as bounded in a lazy language.)

The algorithm is based on the following expansion:

$$\begin{aligned}\pi &= \sum_{i=0}^{\infty} \frac{(i!)^2 2^{i+1}}{(2i+1)!} \\ &= \left(2 + \frac{1}{3} \times\right)\left(2 + \frac{2}{5} \times\right)\left(2 + \frac{3}{7} \times\right) \cdots \left(2 + \frac{i}{2i+1} \times\right) \cdots\end{aligned}$$

A streaming algorithm can convert this infinite sequence of linear fractional transformations (represented as homographies) into an infinite sequence of decimal digits. The consumption operator is matrix multiplication, written $\odot$ in Section 5.1. When a digit n is produced, the state h should be transformed into

$$\left(\begin{smallmatrix} 10 & -10n \\ 0 & 1 \end{smallmatrix}\right) \odot h$$

Any tail of the input sequence represents a value between 3 and 4, so homography h determines the next digit when

$$\lfloor abs\ h\ 3 \rfloor \mathrel{\widehat{=}} \lfloor abs\ h\ 4 \rfloor$$

(in which case, the digit is the common value of these two expressions). This reasoning gives us the following program:

$$pi \mathrel{\widehat{=}} stream\ prod\ (\odot)\ ident\ lfts$$

where

$$\begin{aligned} lfts &\mathrel{\widehat{=}} \left[\left(\begin{smallmatrix} k & 4k+2 \\ 0 & 2k+1 \end{smallmatrix}\right) \mid k \leftarrow [1..]\right] \\ prod\ h &\mathrel{\widehat{=}} \mathbf{if}\ \lfloor abs\ h\ 4 \rfloor == n\ \mathbf{then}\ \mathsf{Just}\ (n, \left(\begin{smallmatrix} 10 & -10n \\ 0 & 1 \end{smallmatrix}\right) \odot h)\ \mathbf{else}\ \mathsf{Nothing} \\ &\qquad \mathbf{where}\ n \mathrel{\widehat{=}} \lfloor abs\ h\ 3 \rfloor \end{aligned}$$

6.2 Arithmetic coding

Arithmetic coding [40] is a method for data compression. It can be more *effective* than rival schemes such as Huffman coding, while still being as *efficient*. Moreover, it is well suited to *adaptive* encoding, in which the coding scheme evolves to match the text being encoded.

The basic idea of arithmetic encoding is simple. The message to be encoded is broken into *symbols*, such as characters, and each symbol of the message is associated with a semi-open *subinterval* of the unit interval $[0\,..\,1)$. Encoding starts with the unit interval, and *narrows* it according to the intervals associated with each symbol of the message in turn. The encoded message is the binary representation of some *fraction* chosen from the final 'target' interval.

In [4] we present a detailed derivation of arithmetic encoding and decoding. We merely outline the development of encoding here, to show where streaming fits in. Decoding follows a similar process to encoding, starting with the unit interval and homing in on the binary fraction, reconstructing the plaintext in the process; but we will not discuss it here.

The encoding process can be captured as follows. The type *Interval* represents intervals of the real line, usually *subunits* (subintervals of the unit interval):

$$\begin{aligned}&\mathit{unit} :: \mathit{Interval}\\ &\mathit{unit} \mathrel{\widehat{=}} [0\,..\,1)\end{aligned}$$

Narrowing an interval lr by a subunit pq yields a subinterval of lr, which stands in the same relation to lr as pq does to *unit*.

$$\begin{aligned}&\mathit{narrow} :: \mathit{Interval} \to \mathit{Interval} \to \mathit{Interval}\\ &\mathit{narrow}\ [l\,..\,r)\ [p\,..\,q) \mathrel{\widehat{=}} [l + (r{-}l) \times p\,..\,l + (r{-}l) \times q)\end{aligned}$$

We consider only non-adaptive encoding here for simplicity: adaptivity turns out to be orthogonal to streaming. We therefore represent each symbol by a fixed interval.

Encoding is a two-stage process: narrowing intervals to a target interval, and generating the binary representation of a fraction within that interval (missing its final 1).

$$\begin{aligned}&\mathit{encode} :: [\mathit{Interval}] \to [\mathit{Bool}]\\ &\mathit{encode} \mathrel{\widehat{=}} \mathsf{unfoldr}\ \mathit{nextBit} \cdot \mathsf{foldl}\ \mathit{narrow}\ \mathit{unit}\end{aligned}$$

where

$$\begin{aligned}&\mathit{nextBit}\ (l, r)\\ &\quad \mid r \le {}^1\!/_2 \quad \mathrel{\widehat{=}} \mathsf{Just}\ (\mathit{False}, \mathit{narrow}\ (0, 2)\ (l, r))\\ &\quad \mid {}^1\!/_2 \le l \quad \mathrel{\widehat{=}} \mathsf{Just}\ (\mathit{True}, \mathit{narrow}\ (-1, 1)\ (l, r))\\ &\quad \mid l < {}^1\!/_2 < r \mathrel{\widehat{=}} \mathsf{Nothing}\end{aligned}$$

This is a metamorphism.

As described, this is not a very efficient encoding method: the entire message has to be digested into a target interval before any of the fraction can be generated. However, the streaming condition holds, and bits of the fraction can be produced before all of the message is consumed:

$$\begin{aligned}&\mathit{encode} \quad :: [\mathit{Interval}] \to [\mathit{Bool}]\\ &\mathit{encode}\ m = \mathit{stream}\ \mathit{nextBit}\ \mathit{narrow}\ \mathit{unit}\end{aligned}$$

7 Future and related work

The notion of metamorphisms in general and of streaming algorithms in particular arose out of our work on arithmetic coding [4]. Since then, we have seen the same principles cropping up in other areas, most notably in the context of various kinds of numeric representations: the radix conversion problem from Section 3.2, continued fractions as described in Section 5, and computations with infinite compositions of homographies as used in Section 6.1. Indeed, one might even see arithmetic coding as a kind of numeric representation problem.

7.1 Generic streaming

Our theory of metamorphisms could easily be generalized to other datatypes: there is nothing to prevent consideration of folds consuming and unfolds producing datatypes other than lists. However, we do not currently have any convincing examples.

Perhaps related to the lack of convincing examples for other datatypes, it is not clear what a generic theory of streaming algorithms would look like. List-oriented streaming relies essentially on foldl, which does not generalize in any straightforward way to other datatypes. (We have in the past attempted to show how to generalize scanl to arbitrary datatypes [9–11], and Pardo [31] has improved on these attempts; but we do not see yet how to apply those constructions here.)

However, the unfold side of streaming algorithms does generalize easily, to certain kinds of datatype if not obviously all of them. Consider producing a data structure of the type

$$\textbf{data}\ \mathit{Generic}\ \tau\ \alpha = \mathit{Gen}\ (\mathsf{Maybe}\ (\alpha, \tau\ (\mathit{Generic}\ \tau\ \alpha)))$$

for some instance τ of the type class *Functor*. (Lists essentially match this pattern, with τ the identity functor. The type Tree of internally-labelled binary trees introduced in Section 2 matches too, with τ being the pairing functor. In general, datatypes of this form have an empty structure, and all non-empty structures consist of a root element and an τ-shaped collection of children.) It is straightforward to generalize the streaming condition to such types:

$$f\ c = \mathsf{Just}\ (b, c') \Rightarrow f\ (g\ c\ a) = \mathsf{Just}\ (b, \mathit{fmap}\ (\lambda u \rightarrow g\ u\ a)\ c')$$

(This has been called an 'τ-invariant' or 'mongruence' [23] elsewhere.) Still, we do not have any useful applications of an unfold to a *Generic* type after a foldl.

7.2 Related work: Back to basics

Some of the ideas presented here appeared much earlier in work of Hutton and Meijer [21]. They studied *representation changers*, consisting of a function followed by the converse of a function. Their representation changers are analogous

to our metamorphisms, with the function corresponding to the fold and the converse of a function to the unfold: in a relational setting, an unfold is just the converse of a fold, and so our metamorphisms could be seen as a special case of representation changers in which both functions are folds. We feel that restricting attention to the special case of folds and unfolds is worthwhile, because we can capitalize on their universal properties; without this restriction, one has to resort to reasoning from first principles.

Hutton and Meijer illustrate with two examples: carry-save incrementing and radix conversion. The carry-save representation of numbers is redundant, using the redundancy to avoid rippling carries. Although incrementing such a number can be seen as a change of representation, it is a rather special one, as the point of the exercise is to copy as much of the input as possible straight to the output; it isn't immediately clear how to fit that constraint into our pattern of folding to an abstract value and independently unfolding to a different representation. Their radix conversion is similar to ours, but their resulting algorithm is not streaming: all of the input must be read before any of the output is produced.

Acknowledgements

The material on arithmetic coding in Section 6.2, and indeed the idea of streaming algorithms in the first place, came out of joint work with Richard Bird [4] and Barney Stratford. The principles behind reversing the order of evaluation and defunctionalization presented in Section 4.2 have been known for a long time [7, 34], but the presentation used here is due to Geraint Jones.

We are grateful to members of the *Algebra of Programming* research group at Oxford and of *IFIP Working Group 2.1*, the participants in the *Datatype-Generic Programming* project, and the anonymous *Mathematics of Program Construction* referees for their helpful suggestions regarding this work.

References

1. Lex Augusteijn. Sorting morphisms. In S. D. Swierstra, P. R. Henriques, and J. N. Oliveira, editors, *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 1–27, 1998.
2. M. Beeler, R. W. Gosper, and R. Schroeppel. Hakmem. AIM 239, MIT, February 1972.
3. Richard Bird and Oege de Moor. *The Algebra of Programming.* Prentice-Hall, 1996.
4. Richard Bird and Jeremy Gibbons. Arithmetic coding with folds and unfolds. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming 4*, volume 2638 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
5. Richard S. Bird. *Introduction to Functional Programming Using Haskell.* Prentice-Hall, 1998.
6. Richard S. Bird and Philip L. Wadler. *An Introduction to Functional Programming.* Prentice-Hall, 1988.

7. Eerke Boiten. The many disguises of accumulation. Technical Report 91-26, Department of Informatics, University of Nijmegen, December 1991.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
9. Jeremy Gibbons. *Algebras for Tree Algorithms*. D. Phil. thesis, Programming Research Group, Oxford University, 1991. Available as Technical Monograph PRG-94. ISBN 0-902928-72-4.
10. Jeremy Gibbons. Polytypic downwards accumulations. In Johan Jeuring, editor, *Proceedings of Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, Marstrand, Sweden, June 1998. Springer-Verlag.
11. Jeremy Gibbons. Generic downwards accumulations. *Science of Computer Programming*, 37:37–65, 2000.
12. Jeremy Gibbons. Calculating functional programs. In Roland Backhouse, Roy Crole, and Jeremy Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*, pages 148–203. Springer-Verlag, 2002.
13. Jeremy Gibbons. Origami programming. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, Cornerstones in Computing. Palgrave, 2003.
14. Jeremy Gibbons. An unbounded spigot algorithm for the digits of π. Draft, November 2003.
15. Jeremy Gibbons and Graham Hutton. Proof methods for corecursive programs. Submitted for publication, March 2004.
16. Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 273–279, Baltimore, Maryland, September 1998.
17. Bill Gosper. Continued fraction arithmetic. Unpublished manuscript, 1981.
18. Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, volume 283 of *Lecture Notes in Computer Science*, pages 140–157. Springer-Verlag, September 1987.
19. C. A. R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.
20. John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, April 1989. Also in [36].
21. Graham Hutton and Erik Meijer. Back to basics: Deriving representation changers functionally. *Journal of Functional Programming*, 6(1):181–188, 1996.
22. M. A. Jackson. *Principles of Program Design*. Academic Press, 1975.
23. Bart Jacobs. Mongruences and cofree coalgebras. In *Algebraic Methodology and Software Technology*, volume 936 of *Lecture Notes in Computer Science*, 1995.
24. Simon Peyton Jones. Arbitrary precision arithmetic using continued fractions. INDRA Working Paper 1530, Dept of CS, University College, London, January 1984.
25. Simon Peyton Jones. *The Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
26. David Lester. Vuillemin's exact real arithmetic. In Rogardt Heldal, Carsten Kehler Holst, and Philip Wadler, editors, *Glasgow Functional Programming Workshop*, pages 225–238, 1991.
27. David Lester. Effective continued fractions. In *Proceedings of the Fifteenth IEEE Arithmetic Conference*, 2001.
28. Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

29. Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.
30. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.
31. Alberto Pardo. Generic accumulations. In Jeremy Gibbons and Johan Jeuring, editors, *Generic Programming*, pages 49–78. Kluwer Academic Publishers, 2003. Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloß Dagstuhl, July 2002. ISBN 1-4020-7374-7.
32. Peter John Potts. *Exact Real Arithmetic using Möbius Transformations*. PhD thesis, Imperial College, London, July 1998.
33. Stanley Rabinowitz and Stan Wagon. A spigot algorithm for the digits of π. *American Mathematical Monthly*, 102(3):195–203, 1995.
34. John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher Order and Symbolic Computing*, 11(4):363–397, 1998. Reprinted from the Proceedings of the 25th ACM National Conference, 1972.
35. Doaitse Swierstra and Oege de Moor. Virtual data structures. In Bernhard Möller, Helmut Partsch, and Steve Schumann, editors, *IFIP TC2/WG2.1 State-of-the-Art Report on Formal Program Development*, volume 755 of *Lecture Notes in Computer Science*, pages 355–371. Springer-Verlag, 1993.
36. David A. Turner, editor. *Research Topics in Functional Programming*. University of Texas at Austin, Addison-Wesley, 1990.
37. Varmo Vene and Tarmo Uustalu. Functional programming with apomorphisms (corecursion). *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics*, 47(3):147–161, 1998. 9th Nordic Workshop on Programming Theory.
38. Jean Vuillemin. Exact real arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8):1087–1105, August 1990.
39. Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.
40. I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.