JFP **23** (2): 161–173, 2013. © Cambridge University Press 2012

doi:10.1017/S0956796812000391 First published online 19 October 2012

FUNCTIONAL PEARLS

Metamorphism in jigsaw

KEISUKE NAKANO

Center for Frontier Science and Engineering, The University of Electro-Communications, Japan
(*e-mail:* `ksk@cs.uec.ac.jp`)

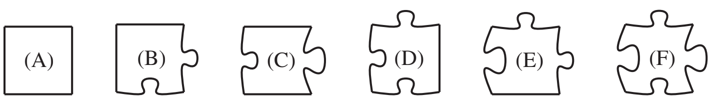
Abstract

A metamorphism is an unfold after a fold, consuming an input by the fold then generating an output by the unfold. It is typically useful for converting data representations, e.g., radix conversion of numbers. (Bird and Gibbons, Lecture Notes in Computer Science, vol. 2638, 2003, pp. 1–26) have shown that metamorphisms can be incrementally processed in *streaming* style when a certain condition holds because part of the output can be determined before the whole input is given. However, whereas radix conversion of fractions is amenable to streaming, radix conversion of natural numbers cannot satisfy the condition because it is impossible to determine part of the output before the whole input is completed. In this paper, we present a *jigsaw* model in which metamorphisms can be partially processed for outputs even when the streaming condition does not hold. We start with how to describe the 3-to-2 radix conversion of natural numbers using our model. The jigsaw model allows us to process metamorphisms in a flexible way that includes parallel computation. We also apply our model to other examples of metamorphisms.

1 Introduction

Consider a problem: 'Convert a given ternary (base-3) number to its binary equivalent'. To solve this, we first compute the number $\sum_{i=0}^{k-1} a_i 3^i$ from a given sequence $\{a_i\}_{i=0}^{k-1}$ and then obtain a sequence $\{b_i\}_{i=0}^{l-1}$ (for some l) through l-fold division by 2.

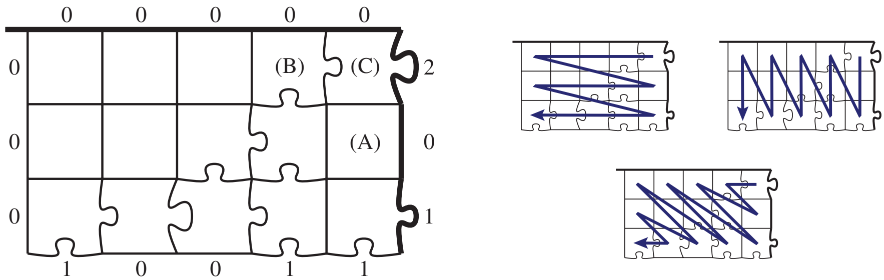
The problem can be solved without using addition, subtraction, multiplication, and division. The solution is given by the following six jigsaw pieces:



Note that there are only three kinds of curves. Each curve represents a digit: , , and  correspond to 0, 1, and 2. We assume that there are infinitely many pieces and pieces may not be rotated. The radix conversion problem is solved by placing these jigsaw pieces at proper positions.

We shall now present how to convert a ternary number to its binary representation by placing these jigsaw pieces. For instance, suppose that a ternary number 201 is given. The conversion is outlined in Figure 1(a). We start with a board that has a

https://doi.org/10.1017/S0956796812000391 Published online by Cambridge University Press



(a) Conversion from 201_3 to 10011_2. (b) Piece placing strategies.

Fig. 1. (Colour online) 3-to-2 radix conversion with jigsaw pieces.

straight edge on the top and an edge with curves representing 2, 0, and 1 (from top to bottom) on the right. Initially, the only piece we can place at the top right corner is (C). Next, two pieces (A) and (B) are placed beneath and to the left of the last one respectively. By repeating the placement of pieces in this way until a straight line appears at the left, the conversion procedure is completed. Finally, the result 10011 appears as the curves of the bottom edge.

It is interesting that many orders are possible to obtain the result. Since there is precisely one piece for each combination of right and top edge, the final result does not depend on the order in which pieces are placed. Jigsaw pieces can be placed horizontally, vertically, or diagonally as shown in Figure 1(b). Of course, one may use other random or elaborated strategies. This property naturally contributes to parallelizing the computation.

We will formalize this jigsaw procedure in functional style in this paper. Our jigsaw radix conversion can be generalized to *metamorphisms*.

2 The trick revealed

We shall first reveal the trick underlying the jigsaw radix conversion by using the example in Figure 1.

Let $\blacktriangledown$ and $\blacktriangleright$ be left-associative infix operators such that $x \blacktriangledown y = 3x + y$ and $x \blacktriangleright y = 2x + y$. Using these operators, we find $201_3 = 10011_2 = 19$ by $2 \blacktriangledown 0 \blacktriangledown 1 = 1 \blacktriangleright 0 \blacktriangleright 0 \blacktriangleright 1 \blacktriangleright 1 = 19$. We show that the procedure for placing jigsaw pieces achieves symbolic conversion from $2 \blacktriangledown 0 \blacktriangledown 1$ into $1 \blacktriangleright 0 \blacktriangleright 0 \blacktriangleright 1 \blacktriangleright 1$. Watch the change in the boundary between the placed and unplaced areas. Initially the boundary consists of straight horizontal curves $0, 0, 0, 0, 0$, and vertical curves representing $2, 0, 1$. We represent the boundary by $0 \blacktriangleright 0 \blacktriangleright 0 \blacktriangleright 0 \blacktriangleright 0 \blacktriangledown 2 \blacktriangledown 0 \blacktriangledown 1$ using $\blacktriangleright$ and $\blacktriangledown$. By placing the (C) piece, the boundary is changed into $0 \blacktriangleright 0 \blacktriangleright 0 \blacktriangleright 0 \blacktriangledown 1 \blacktriangleright 0 \blacktriangledown 0 \blacktriangledown 1$. Piece placing corresponds in general to an update of the boundary $\ldots \blacktriangleright T \blacktriangledown R \ldots$ into $\ldots \blacktriangledown L \blacktriangleright B \ldots$ where T, R, L, and B correspond to curves at the top, right, left, and bottom of the piece. The key to jigsaw radix conversion is that every jigsaw piece satisfies an equation $x \blacktriangleright T \blacktriangledown R = x \blacktriangledown L \blacktriangleright B$ for any x, e.g., $x \blacktriangleright 1 \blacktriangledown 1 = x \blacktriangledown 2 \blacktriangleright 0 = 6x + 4$ for the (E) piece. This implies that the 'value'

https://doi.org/10.1017/S0956796812000391 Published online by Cambridge University Press

of the boundary is invariant when pieces are placed. Therefore, we have

$$0 \blacktriangleright 0 \blacktriangleright 0 \blacktriangleright 0 \blacktriangleright 0 \blacktriangledown 2 \blacktriangledown 0 \blacktriangledown 1 = 0 \blacktriangledown 0 \blacktriangledown 0 \blacktriangledown 1 \blacktriangleright 0 \blacktriangleright 0 \blacktriangleright 1 \blacktriangleright 1$$

where the left- and right-hand sides correspond to the boundary at the initial and final boards. Since $(0\,\blacktriangleright)$ and $(0\,\blacktriangledown)$ are identity, $2 \blacktriangledown 0 \blacktriangledown 1 = 1 \blacktriangleright 0 \blacktriangleright 0 \blacktriangleright 1 \blacktriangleright 1$ holds.

While we have started with five 0s for the top edge in the above example, we generally cannot predict how many 0s will be needed. We may have to add $(0\,\blacktriangleright)$s at the head of the boundary on demand and stop the procedure when (i) no $\blacktriangleright$ occurs at the left of $\blacktriangledown$ and (ii) only 0s occur at the left of $\blacktriangledown$. Condition (i) means that no piece can be placed, and condition (ii) means that a straight line appears at the left of the board. The procedure always terminates because the value of the left operand of the rightmost $\blacktriangledown$ is reduced by boundary updating. The occurrence of $(0\,\blacktriangledown)$ at the head of the boundary can be eliminated at any time because it is an identity.

3 Formalization of a jigsaw model

We formalize our jigsaw procedure in functional style. We will make use of a Haskell-like notation for familiarity.

Let V and H be types of curves on vertical and horizontal edges. Our jigsaw procedure specifies a function of $[\mathsf{V}] \to [\mathsf{H}]$. A jigsaw procedure is characterized by the following three factors: a set of jigsaw pieces, a sequence of curves at the top edge of the initial board, and a sequence of curves at the left edge of the final board.

First, a set of jigsaw pieces is given by a total function

$$\mathit{pieces} :: (\mathsf{V}, \mathsf{H}) \to (\mathsf{H}, \mathsf{V})$$

which determines curves on its left and bottom edges from those on its right and top edges. For example, the set of jigsaw pieces (A) to (F) presented in Section 1 is given by

```
data V = V0 | V1 | V2  -- 0, 1 and 2 as ternary digits
data H = H0 | H1       -- 0 and 1 as binary digits
pieces (V0,H0) = (H0,V0); pieces (V1,H0) = (H1,V0); pieces (V2,H0) = (H0,V1)
pieces (V0,H1) = (H1,V1); pieces (V1,H1) = (H0,V2); pieces (V2,H1) = (H1,V2)
```

where V_0, V_1, V_2, H_0, and H_1 correspond to curves representing 0, 1, 2, 0, and 1.

Second, a sequence of curves at the top edge of the initial board is specified by a single horizontal curve $h_0 :: \mathsf{H}$. We assume that the top edge has an infinite repetition of the h_0 curve. In our 3-to-2 radix conversion, the curve is given by H_0 so that the top of the initial board forms an infinite straight horizontal line.

Third, a sequence of curves at the left edge of the final board is specified by a single vertical curve $v_0 :: \mathsf{V}$. Our jigsaw procedure terminates when all the vertical curves become the v_0 curve. In our 3-to-2 radix conversion, the curve is given by V_0 so that we complete the procedure when a straight vertical line appears at the left of the final board.

Therefore, our jigsaw procedure should be given by a function

$$\mathit{jigsaw} :: ((\mathsf{V}, \mathsf{H}) \to (\mathsf{H}, \mathsf{V})) \to \mathsf{H} \to \mathsf{V} \to [\mathsf{V}] \to [\mathsf{H}]$$

https://doi.org/10.1017/S0956796812000391 Published online by Cambridge University Press

which takes the above three factors and returns a function over lists. We assume that lists of horizontal curves are aligned in a *right-to-left* manner, and those of vertical curves are aligned in a *bottom-to-top* manner. Thereby Figure 1 shows that the *jigsaw* function transforms $[\mathsf{V}_1, \mathsf{V}_0, \mathsf{V}_2]$ into $[\mathsf{H}_1, \mathsf{H}_1, \mathsf{H}_0, \mathsf{H}_0, \mathsf{H}_1]$.

We assume that three arguments of the *jigsaw* function, $\mathit{pieces} :: (\mathsf{V}, \mathsf{H}) \to (\mathsf{H}, \mathsf{V})$, $h_0 :: \mathsf{H}$, and $v_0 :: \mathsf{V}$, satisfy a requirement that

$$\mathit{pieces}\ (v_0, h_0) = (h_0, v_0). \tag{1}$$

This guarantees that only terminating curves can occur at the left of terminating curves under the top horizontal edge.

4 Definition of the jigsaw function

We implement the *jigsaw* function based on the idea of 'boundary updating', explained in Section 2. A boundary is represented by a list of either vertical or horizontal curves from bottom right to top left (right-to-left in (▼, ►)-representation in Section 2), which has type $[\mathsf{Edge}\ a\ b]$ with

$$\mathbf{data}\ \mathsf{Edge}\ a\ b\ =\ \mathsf{Vert}\ a\ \mid\ \mathsf{Horiz}\ b$$

where the types of vertical and horizontal curves are parameterized. The Edge type is isomorphic to the Either type in Haskell's standard library. We do not use the Either type, however, because its constructors are Left and Right, which may cause confusion with the left and right of jigsaw pieces. The initial boundary only consists of vertical edges from the bottom to the top, i.e., $[\mathsf{Vert}\ x_0, \ldots, \mathsf{Vert}\ x_{l-1}]$, and a horizontal edge $\mathsf{Horiz}\ h_0$ will be added on demand at the tail of the list with a curve h_0 of the top edge of the board.

We implement the *jigsaw* function as follows:

$$\begin{array}{l}
\mathit{jigsaw}\ \mathit{pieces}\ h_0\ v_0\ =\ \mathit{map}\ \mathit{unHoriz}\ \cdot\ \mathit{place}\ \cdot\ \mathit{map}\ \mathsf{Vert} \\
\quad \mathbf{where}\ \begin{array}[t]{lcl}
\mathit{unHoriz}\ (\mathsf{Horiz}\ x) & = & x \\
\mathit{isHoriz}\ (\mathsf{Horiz}\ _) & = & \mathsf{True} \\
\mathit{isHoriz}\ (\mathsf{Vert}\ _) & = & \mathsf{False} \\
\mathit{place}\ =\ \mathit{until}\ (\mathit{all}\ \mathit{isHoriz})\ (\mathit{step}\ \mathit{pieces}\ h_0\ v_0).
\end{array}
\end{array}$$

Here, $\mathit{place} :: [\mathsf{Edge}\ a\ b] \to [\mathsf{Edge}\ a\ b]$ repeatedly updates the boundary, preserving its value, until it contains only Horiz elements; the loop function *until* is defined by

$$\begin{array}{l}
\mathit{until}\ :: (a \to \mathsf{Bool}) \to (a \to a) \to (a \to a) \\
\mathit{until}\ p\ f\ x\ \mid\ p\ x\ \ \ =\ x \\
\qquad\qquad\quad \mid\ \neg p\ x\ =\ \mathit{until}\ p\ f\ (f\ x).
\end{array}$$

The auxiliary function $\mathit{unHoriz} :: \mathsf{Edge}\ a\ b \to b$ unwraps the Horiz constructor; although it is a partial function, it is clear that *place* always returns a list that contains no Vert elements. For the function *step*, it suffices to make a single value-preserving boundary update, assuming that at least one Vert element is present, and

https://doi.org/10.1017/S0956796812000391 Published online by Cambridge University Press

guaranteeing to make progress. We specify it as follows:

$$
\begin{aligned}
&\mathit{step}\ p\ h_0\ v_0\ (xs ⧺ [\mathsf{Vert}\ v, \mathsf{Horiz}\ h] ⧺ ys) \ = \ xs ⧺ [\mathsf{Horiz}\ h', \mathsf{Vert}\ v'] ⧺ ys \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where}\ (h', v') = \mathit{pieces}\ (v, h) \\
&\mathit{step}\ p\ h_0\ v_0\ (xs ⧺ [\mathsf{Vert}\ v]) \mid\ v = v_0 = \ xs \\
&\qquad\qquad\qquad\qquad\qquad\quad\ \mid\ v \neq v_0 = \ xs ⧺ [\mathsf{Vert}\ v, \mathsf{Horiz}\ h_0].
\end{aligned}
$$

The first rule of the *step* function corresponds to placing a single piece. The second rule represents that a terminating curve at the left of the board does not contribute to the result because of requirement (1). The third rule corresponds to board extension with the h_0 edge.

Note that the patterns in the specification of *step* may match multiple places, but the computation is still confluent because no pairs of redexes interfere with one another when the definition is regarded as a rewriting system. This is a trivial case of confluent rewriting systems (Baader & Nipkow 1998). The termination of *place* depends on the three parameters: *pieces*, h_0, and v_0. When *pieces* is given by $\mathit{pieces}\ (x, y) = (y, v)$ with a constant $v \neq v_0$ for any x and y, the computation of $\mathit{place}\ [v]$ will generate an infinite number of h_0s.

Nondeterminism of pattern matching for the *step* function corresponds to flexibility in the order of placing pieces. When we always rewrite the rightmost occurrence of $[\mathsf{Vert}\ v, \mathsf{Horiz}\ h]$ at the input (extending it with $\mathsf{Horiz}\ h_0$ if possible), the procedure corresponds to the horizontal placement of pieces. When we always rewrite the leftmost occurrence, the procedure corresponds to the vertical placement of pieces. When we simultaneously rewrite multiple occurrences of $[\mathsf{Vert}\ v, \mathsf{Horiz}\ h]$, the procedure corresponds to the parallel placement of pieces.

Using the *jigsaw* function, the 3-to-2 radix conversion $\mathit{radixConv}_{3,2}$ is implemented by

$$
\begin{aligned}
&\mathit{radixConv}_{3,2} \ :: \ [\mathsf{V}] \to [\mathsf{H}] \\
&\mathit{radixConv}_{3,2} \ = \mathit{jigsaw}\ \mathit{pieces}\ \mathsf{H}_0\ \mathsf{V}_0
\end{aligned}
$$

where V, H, V_0, H_0, and *pieces* are given as examples in Section 3.

5 Metamorphism in the jigsaw model

We start with an ordinary definition of metamorphism, which is an unfold after a fold, consuming the input by the fold, and generating an output by the unfold. In this paper we use the following variations of the *foldr* and *unfoldr* functions:

$$
\begin{aligned}
&\mathit{foldr} :: ((b, a) \to a) \to a \to [b] \to a \\
&\mathit{foldr}\ g\ e\ [\,] \qquad = e \\
&\mathit{foldr}\ g\ e\ (x : xs) \ = \ g\ (x,\ \mathit{foldr}\ g\ e\ xs) \\
&\mathit{unfoldr} :: (a \to \mathsf{Maybe}(b, a)) \to a \to [b] \\
&\mathit{unfoldr}\ f\ s \ = \ \textbf{case}\ f\ s\ \textbf{of}\ \mathsf{Nothing} \quad \to [\,] \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{Just}(x, s') \ \to \ x : \mathit{unfoldr}\ f\ s'
\end{aligned}
$$

where the first argument of *foldr* has an uncurried form for convenience of formalization. The 3-to-2 radix conversion $\mathit{radixConv}_{3,2}$ is implemented as a metamorphism

https://doi.org/10.1017/S0956796812000391 Published online by Cambridge University Press

by

$$
\begin{aligned}
&\mathit{radixConv}_{3,2} \;::\; [\mathsf{V}] \to [\mathsf{H}] \\
&\mathit{radixConv}_{3,2} \;=\; \mathit{unfoldr}\ \mathit{modDiv}_2 \cdot \mathit{foldr}\ \mathit{sumMul}_3\ 0 \\
&\quad \textbf{where}\ \mathit{sumMul}_3 :: (\mathsf{V}, \mathit{Int}) \to \mathit{Int} \\
&\qquad\quad \mathit{sumMul}_3\ (\mathsf{V}_i, s) \;=\; i + s \times 3 \\
&\qquad\quad \mathit{modDiv}_2 :: \mathit{Int} \to \mathsf{Maybe}(\mathsf{H}, \mathit{Int}) \\
&\qquad\quad \mathit{modDiv}_2\ s \mid s = 0 \;=\; \mathsf{Nothing} \\
&\qquad\qquad\qquad\quad\;\; \mid s \neq 0 \;=\; \mathsf{Just}(\mathsf{H}_j, t) \quad \textbf{where} \quad j + t \times 2 = s
\end{aligned}
$$

where V and H are defined as in Section 3, and we employ a 'smart' **where**-clause computing remainders for readability. We assume that ternary and binary representations are given by a list of digits from the least significant to the most significant digits. This definition can be easily generalized to m-to-n radix conversion.

As we have shown, the $\mathit{radixConv}_{3,2}$ function can also be implemented in jigsaw style. An important difference from the ordinary definition is that the computation does not involve types (like Int) other than that of inputs and outputs (like V and H). In the ordinary definition, we have to first complete the computation with *foldr* to obtain the integer representation of the input. Even if lazy evaluation is used, *unfoldr* does not produce anything before *foldr* is completed because the first element of the output, which is the least significant digit, is determined by the whole input. On the other hand, the *jigsaw* procedure can start some parallelizable computation when the input is being read.

We will demonstrate that the *jigsaw* function can represent a metamorphism 'under a certain condition' as indicated by the equation

$$
\mathit{unfoldr}\ f \cdot \mathit{foldr}\ g\ e \;=\; \mathit{jigsaw}\ \mathit{pieces}\ h_0\ v_0 \tag{2}
$$

with some *pieces*, h_0, and v_0, which are determined by f, g, and e. The condition is given in the following definition, which will be required to make Equation (2) hold.

Definition 5.1
Three functions $f :: a \to \mathsf{Maybe}(\mathsf{H}, a)$, $g :: (\mathsf{V}, a) \to a$, and $e :: a$ are said to satisfy the *jigsaw condition* with $\mathit{pieces} :: (\mathsf{V}, \mathsf{H}) \to (\mathsf{H}, \mathsf{V})$, $h_0 : \mathsf{H}$, and $v_0 :: \mathsf{V}$ when all of the following clauses are satisfied:

(i) $f\ e = \mathsf{Nothing}$ holds;
(ii) $f(g(v, s)) = \mathsf{Nothing}$ for $(v, s) :: (\mathsf{V}, a)$ if and only if $v = v_0$ and $f\ s = \mathsf{Nothing}$;
(iii) $f^{\#}(g(v, s)) = (h', g(v', s'))$ for $(v, s) :: (\mathsf{V}, a)$ when $(h, s') = f^{\#}\ s$ and $(h', v') = \mathit{pieces}\ (v, h)$, where

$$
\begin{aligned}
&f^{\#} :: a \to (\mathsf{H}, a) \\
&f^{\#}\ s \;=\; \textbf{case}\ f\ s\ \textbf{of}\ \mathsf{Nothing} \;\;\to (h_0, s) \\
&\qquad\qquad\qquad\qquad\quad \mathsf{Just}(h, s') \to (h, s').
\end{aligned}
$$

The variables *pieces*, h_0, and v_0 in the above definition are the arguments of the *jigsaw* function in Equation (2). Condition (i) forces the metamorphism to generate an empty list for the empty list. This restriction is essential to make Equation (2) hold. Condition (ii) stipulates a property on the nonproductive seeds of *unfoldr*.

https://doi.org/10.1017/S0956796812000391 Published online by Cambridge University Press

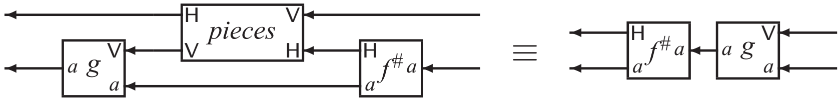


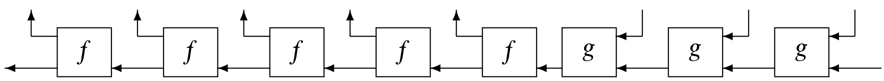
Fig. 2. Jigsaw condition (iii).

Condition (iii) is the most weighty condition, which is visualized as a diagram in Figure 2. Intuitively, this means that our jigsaw procedure can process the f generator using *pieces* before folding the whole input with g. The 'if' part of condition (ii) always holds when condition (iii) holds under requirement (1).

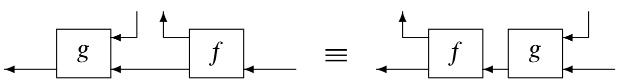
Let us check whether the jigsaw condition is satisfied for two implementations of $radixConv_{3,2}$. Condition (i) holds because $modDiv_2\ 0 = \mathsf{Nothing}$ from the definition. Condition (ii) holds because $modDiv_2(sumMul_3(\mathsf{V}_i, s)) = modDiv_2(i{+}s{\times}3) = \mathsf{Nothing}$ if and only if $i = s = 0$. Condition (iii) holds because

$$
\begin{array}{cl}
 & modDiv_2^{\#}(sumMul_3(\mathsf{V}_i, s)) \\
= & \quad \{\ sumMul_3\ \} \\
 & modDiv_2^{\#}(i + s \times 3) \\
= & \quad \{\ \text{unfold } modDiv_2^{\#} \text{ and merge the branches}\ \} \\
 & (\mathsf{H}_j, t)\ \textbf{where}\ j + t \times 2 = i + s \times 3 \\
= & \quad \{\ \text{division of } t \text{ and } s \text{ by 3 and 2, respectively}\ \} \\
 & (\mathsf{H}_j, t)\ \textbf{where}\ j + i' \times 2 + s' \times 6 = i + j' \times 3 + t' \times 6 \\
 & \qquad\qquad\quad i' + s' \times 3 = t\ \textbf{and}\ j' + t' \times 2 = s \\
= & \quad \{\ s' = t' \text{ from } i, i' \leqslant 2,\ j, j' \leqslant 1 \text{ and uniqueness of division by 6}\ \} \\
 & (\mathsf{H}_j, t)\ \textbf{where}\ j + i' \times 2 = i + j' \times 3\ \textbf{and}\ i' + t' \times 3 = t\ \textbf{and}\ j' + t' \times 2 = s \\
= & \quad \{\ pieces,\ sumMul_3 \text{ and } modDiv_2^{\#}\ \} \\
 & (\mathsf{H}_j, sumMul_3(\mathsf{V}_{i'}, t'))\ \textbf{where}\ (\mathsf{H}_j, \mathsf{V}_{i'}) = pieces\ (\mathsf{V}_i, \mathsf{H}_{j'})\ \textbf{and}\ (j', t') = modDiv_2^{\#}\ s.
\end{array}
$$

Before showing a strict relationship between the jigsaw condition and Equation (2), we give an intuitive explanation for this condition on metamorphisms by comparing with the *streaming condition* (Bird and Gibbons, 2003; Gibbons, 2007). A direct implementation of a metamorphism defined by $unfoldr\ f \cdot foldr\ g :: [\mathsf{V}] \rightarrow [\mathsf{H}]$ works as follows:



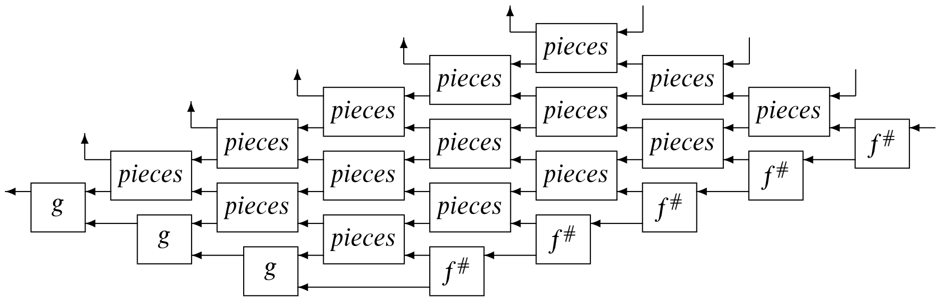
where any part of the output cannot be obtained before reading the whole input. Roughly speaking, the streaming condition enables the order of f and g to alternate like



for 'productive inputs' (x satisfying $f\ x \neq \mathsf{Nothing}$), thereby writing a part of the output after reading only a part of the input. This alternation cannot always happen, of course. It is impossible to do that if we cannot determine the first part of output

https://doi.org/10.1017/S0956796812000391 Published online by Cambridge University Press

before reading the whole input like the radix conversion. On the other hand, jigsaw condition (iii) can alternate the order even when the streaming condition does not hold. Assuming the existence of the *pieces* function satisfying the condition, we can obtain



by using $f^{\#}$ instead of f and repeatedly applying the equivalence relation on *pieces* to the diagram of metamorphism above as many times as possible. The obtained diagram exactly illustrates our jigsaw computational model, in which each *pieces* box represents a jigsaw piece. The leftmost and rightmost *pieces* boxes correspond to jigsaw pieces at the top right and bottom left corners of the board respectively. We can observe that vertical curves are given as an input and horizontal curves are obtained as an output.

Now let us show a strict relationship between our jigsaw model and a metamorphism. We show that Equation (2) holds when the jigsaw condition is satisfied. Since the result of the *place* function does not depend on piece placing strategy, it suffices to prove Equation (2) for a specific strategy. We adopt a horizontal-first piece placing strategy. The proof consists of two steps: First we show that the horizontal piece placing corresponds to the behavior of the *unfoldr* function; then we show that its vertical repetition corresponds to the behavior of the *foldr* function. We start with the lemma for the case where the *place* function takes a boundary containing only one vertical edge (with Vert) at its head. We use the following equations on the *place* function:

$$\begin{aligned} \mathit{place}\ \mathit{xs} &= \mathit{place}\ (\mathit{step}\ \mathit{pieces}\ h_0\ v_0\ \mathit{xs}) && (3)\\ \mathit{place}\ (\mathsf{Horiz}\ h : \mathit{xs}) &= \mathsf{Horiz}\ h : \mathit{place}\ \mathit{xs} && (4) \end{aligned}$$

which are obvious from the definition of *place* and *step*. In the rest of this section, we will assume that *pieces*, h_0, and v_0 are fixed, thus so are *place* and $f^{\#}$.

Lemma 5.1
Let f, g, and e satisfy the jigsaw condition with *pieces*, h_0, and v_0. For all v and s,

$$\mathit{place}\ (\mathsf{Vert}\ v : \mathit{map}\ \mathsf{Horiz}\ (\mathit{unfoldr}\ f\ s)) = \mathit{map}\ \mathsf{Horiz}\ (\mathit{unfoldr}\ (f \ltimes g)\ (v, s))$$

where the $f \ltimes g$ function is defined by

$$\begin{aligned} &(f \ltimes g) :: (\mathsf{V}, a) \to \mathsf{Maybe}(\mathsf{H}, (\mathsf{V}, a))\\ &(f \ltimes g)\ (v, s) = \mathbf{if}\ f(g(v, s)) = \mathsf{Nothing}\ \mathbf{then}\ \mathsf{Nothing}\\ &\qquad\qquad\qquad \mathbf{else}\ \mathsf{Just}(h', (v', s'))\ \mathbf{where}\ (h, s') = f^{\#}\ s\ \mathbf{and}\ (h', v') = \mathit{pieces}\ (v, h). \end{aligned}$$

https://doi.org/10.1017/S0956796812000391 Published online by Cambridge University Press

Proof
We prove the statement by coinduction.

$$
\begin{array}{rl}
& place\ (\mathsf{Vert}\ v : map\ \mathsf{Horiz}\ (unfoldr\ f\ s)) \\
= & \quad \{\ unfoldr \text{ and } map\ \} \\
& \textbf{case}\ f\ s\ \textbf{of}\ \begin{array}[t]{ll} \mathsf{Nothing} & \rightarrow place\ [\mathsf{Vert}\ v] \\ \mathsf{Just}(h, s') & \rightarrow place\ (\mathsf{Vert}\ v : \mathsf{Horiz}\ h : map\ \mathsf{Horiz}\ (unfoldr\ f\ s')) \end{array} \\
= & \quad \{\ \text{equation (3) and } step \text{ for } [\mathsf{Vert}\ v]\ \} \\
& \textbf{case}\ f\ s\ \textbf{of}\ \begin{array}[t]{ll} \mathsf{Nothing} & \rightarrow \textbf{if}\ v = v_0\ \textbf{then}\ [\,]\ \textbf{else}\ place\ [\mathsf{Vert}\ v, \mathsf{Horiz}\ h_0] \\ \mathsf{Just}(h, s') & \rightarrow place\ (\mathsf{Vert}\ v : \mathsf{Horiz}\ h : map\ \mathsf{Horiz}\ (unfoldr\ f\ s')) \end{array} \\
= & \quad \{\ \text{merge the } \textbf{else} \text{ clause in the } \mathsf{Nothing} \text{ branch and the } \mathsf{Just} \text{ branch using } f^{\#}\ \} \\
& \textbf{if}\ f\ s = \mathsf{Nothing}\ \wedge\ v = v_0\ \textbf{then}\ [\,] \\
& \textbf{else}\ place\ (\mathsf{Vert}\ v : \mathsf{Horiz}\ h : map\ \mathsf{Horiz}\ (unfoldr\ f\ s'))\quad \textbf{where}\ (h, s') = f^{\#}\ s \\
= & \quad \{\ \text{equation (3) and } step\ \} \\
& \textbf{if}\ f\ s = \mathsf{Nothing}\ \wedge\ v = v_0\ \textbf{then}\ [\,] \\
& \textbf{else}\ place\ (\mathsf{Horiz}\ h' : \mathsf{Vert}\ v' : map\ \mathsf{Horiz}\ (unfoldr\ f\ s')) \\
& \qquad \textbf{where}\ (h, s') = f^{\#}\ s\ \textbf{and}\ (h', v') = pieces\ (v, h) \\
= & \quad \{\ \text{equation (4)}\ \} \\
& \textbf{if}\ f\ s = \mathsf{Nothing}\ \wedge\ v = v_0\ \textbf{then}\ [\,] \\
& \textbf{else}\ \mathsf{Horiz}\ h' : place\ (\mathsf{Vert}\ v' : map\ \mathsf{Horiz}\ (unfoldr\ f\ s')) \\
& \qquad \textbf{where}\ (h, s') = f^{\#}\ s\ \textbf{and}\ (h', v') = pieces\ (v, h) \\
= & \quad \{\ \text{coinduction principle}\ \} \\
& \textbf{if}\ f\ s = \mathsf{Nothing}\ \wedge\ v = v_0\ \textbf{then}\ [\,] \\
& \textbf{else}\ \mathsf{Horiz}\ h' : map\ \mathsf{Horiz}\ (unfoldr\ (f \ltimes g)\ (v', s')) \\
& \qquad \textbf{where}\ (h, s') = f^{\#}\ s\ \textbf{and}\ (h', v') = pieces\ (v, h) \\
= & \quad \{\ map\ \} \\
& map\ \mathsf{Horiz}\ (\textbf{if}\ f\ s = \mathsf{Nothing}\ \wedge\ v = v_0\ \textbf{then}\ [\,]\ \textbf{else}\ h' : unfoldr\ (f \ltimes g)\ (v', s')) \\
& \qquad \textbf{where}\ (h, s') = f^{\#}\ s\ \textbf{and}\ (h', v') = pieces\ (v, h) \\
= & \quad \{\ unfoldr \text{ and jigsaw condition (ii)}\ \} \\
& map\ \mathsf{Horiz}\ (unfoldr\ (f \ltimes g)\ (v, s)) \qquad \square
\end{array}
$$

The above proof employs coinduction because of the coinductivity of the results of *place* and *unfoldr*. Assuming that $unfoldr\ f\ s$ always generates finite lists for any s, we might use induction on the structure of the lists to prove the statement.

Next, we present the relationship between the *place* function and metamorphism, which is described in the following lemma. In the proof of the lemma, we apply an *unfoldr* fusion law (Meijer et al., 1991). For $f :: b \rightarrow (a, b)$, $h :: c \rightarrow (a, c)$, and $g :: c \rightarrow b$,

$$
f\ (g\ z) = \textbf{case}\ h\ z\ \textbf{of}\ \begin{array}[t]{ll} \mathsf{Nothing} & \rightarrow \mathsf{Nothing} \\ \mathsf{Just}(x, y) & \rightarrow \mathsf{Just}(x, g\ y) \end{array} \Longrightarrow unfoldr\ f\ \cdot\ g = unfoldr\ h. \quad (5)
$$

We can simply confirm that f, g, and $h = f \ltimes g$ introduced in Lemma 5.1 satisfy the above fusion condition under jigsaw condition (iii). In addition, we use the following equation on the *place* function:

$$
place\ (xs +\!\!+ ys) = place\ (xs +\!\!+ place\ ys) \qquad (6)
$$

for all *xs* and *ys*, which can be shown by induction on the computation of *place ys*.

Lemma 5.2
Let f, g, and e satisfy the jigsaw condition with *pieces*, h_0, and v_0. For all *vs*,

$$\mathit{place}\ (\mathit{map}\ \mathsf{Vert}\ \mathit{vs}) \ = \ \mathit{map}\ \mathsf{Horiz}\ (\mathit{unfoldr}\ f\ (\mathit{foldr}\ g\ e\ \mathit{vs})).$$

Proof
We prove the statement by induction on *vs*. When $vs = [\,]$, both sides of the equation become $[\,]$ because of jigsaw condition (i). When $vs = w : ws$,

$$\begin{array}{cl}
& \mathit{place}\ (\mathit{map}\ \mathsf{Vert}\ (w : ws)) \\
= & \quad \{\ \mathit{map} \text{ and equation (6) }\} \\
& \mathit{place}\ (\mathsf{Vert}\ w : \mathit{place}\ (\mathit{map}\ \mathsf{Vert}\ ws)) \\
= & \quad \{\text{ induction hypothesis }\} \\
& \mathit{place}\ (\mathsf{Vert}\ w : \mathit{map}\ \mathsf{Horiz}\ (\mathit{unfoldr}\ f\ (\mathit{foldr}\ g\ e\ ws))) \\
= & \quad \{\text{ Lemma 5.1 }\} \\
& \mathit{map}\ \mathsf{Horiz}\ (\mathit{unfoldr}\ (f \ltimes g)\ (w,\ \mathit{foldr}\ g\ e\ ws)) \\
= & \quad \{\text{ fusion law (5) with jigsaw condition (iii) }\} \\
& \mathit{map}\ \mathsf{Horiz}\ (\mathit{unfoldr}\ f\ (g\ (w,\ \mathit{foldr}\ g\ e\ ws))) \\
= & \quad \{\ \mathit{foldr}\ \} \\
& \mathit{map}\ \mathsf{Horiz}\ (\mathit{unfoldr}\ f\ (\mathit{foldr}\ g\ e\ (w : ws)))
\end{array}$$

□

This lemma immediately concludes Equation (2).

Theorem 5.1
Let f, g, and e satisfy the jigsaw condition with *pieces*, h_0, and v_0. For all lists *vs*,

$$\mathit{jigsaw}\ \mathit{pieces}\ h_0\ v_0\ vs = \mathit{unfoldr}\ f\ (\mathit{foldr}\ g\ e\ vs).$$

Proof

$$\begin{array}{rcl}
\mathit{jigsaw}\ \mathit{pieces}\ h_0\ v_0\ vs & = & \quad \{\ \mathit{jigsaw}\ \} \\
& & \mathit{map}\ \mathit{unHoriz}\ (\mathit{place}\ (\mathit{map}\ \mathsf{Vert}\ vs)) \\
& = & \quad \{\text{ Lemma 5.2 }\} \\
& & \mathit{map}\ \mathit{unHoriz}\ (\mathit{map}\ \mathsf{Horiz}\ (\mathit{unfoldr}\ f\ (\mathit{foldr}\ g\ e\ vs))) \\
& = & \quad \{\ \mathit{unHoriz} \text{ and } \mathit{map}\ \} \\
& & \mathit{unfoldr}\ f\ (\mathit{foldr}\ g\ e\ vs)
\end{array}$$

□

6 Applications

We have shown how to implement *m*-to-*n* radix conversion in the jigsaw model. This section presents two other examples of metamorphism that can be implemented with the jigsaw model. The jigsaw condition requires us to discover an appropriate function *pieces*. Although it is difficult to show a general method for constructing the function, the following examples may help us to find candidates of the *pieces* function.

https://doi.org/10.1017/S0956796812000391 Published online by Cambridge University Press

Group-by procedure for multiple data lists. 'Group by' is a core database operation that groups an input data list into lists of data having the same key. We assume that an input is given as a divided list, so we concatenate them to obtain a list of all data. This operation is a metamorphism consisting of concatenation and grouping. When k and v are types of keys and values, both V and H are $[(k,v)]$, which is the same type as intermediate data.

$$
\begin{array}{l}
\mathit{flipApp} :: ([(k,v)],[(k,v)]) \to [(k,v)] \\
\mathit{flipApp}\ (l_1,l_2) \ = \ l_2 + l_1 \\
\mathit{partByHd} :: [(k,v)] \to \mathsf{Maybe}([(k,v)],[(k,v)]) \\
\mathit{partByHd}\ [\,] \qquad\quad = \ \mathsf{Nothing} \\
\mathit{partByHd}\ ((k,v):l) \ = \ \mathsf{Just}((k,v):ts,\ fs) \ \ \mathbf{where}\ \ (ts,\ fs) = \mathit{partByKey}\ k\ l \\
\mathit{groupByFst} \ = \ \mathit{unfoldr}\ \mathit{partByHd}\ \cdot\ \mathit{foldr}\ \mathit{flipApp}\ [\,]
\end{array}
$$

where the *partByKey* function takes a key and a list of key-value pairs and returns a pair of lists: the first list contains those elements whose key is equal to the given one; the second list contains the remaining elements. Jigsaw condition (i) holds from the definition of *partByHd*. To make the other jigsaw conditions hold, we use *flipApp*, which appends two lists in flipped order even though we could use a simple append to implement the 'group by' operation. Taking $h_0 = v_0 = [\,]$ and

$$
\begin{array}{lcl}
\mathit{pieces}\ ([\,],[\,]) & = & ([\,],[\,]) \\
\mathit{pieces}\ ([\,],(k,v):l) & = & ((k,v):ts,\ fs) \ \ \mathbf{where}\ \ (ts,\ fs) = \mathit{partByKey}\ k\ l \\
\mathit{pieces}\ ((k,v):l_1,l_2) & = & ((k,v):ts,\ fs) \ \ \mathbf{where}\ \ (ts,\ fs) = \mathit{partByKey}\ k\ (l_1 + l_2),
\end{array}
$$

we have $\mathit{groupByFst} = \mathit{jigsaw}\ \mathit{pieces}\ h_0\ v_0$. Jigsaw condition (ii) can be confirmed with a simple calculation. For condition (iii) with $v = l_1$ and $s = (k,v) : l_2$

$$
\begin{array}{cl}
 & \mathit{partByHd}^{\#}(\mathit{flipApp}(l_1,(k,v):l_2)) \\
= & \quad \{\ \mathit{flipApp} \text{ and } \mathit{partByHd}^{\#}\ \} \\
 & ((k,v):ts,\ fs) \ \ \mathbf{where}\ \ (ts,\ fs) = \mathit{partByKey}\ k\ (l_2 + l_1) \\
= & \quad \{\ \text{properties of } \mathit{partByKey}\ \} \\
 & (ts',\ fs_2 + fs_1) \ \ \mathbf{where}\ (ts',\ fs_1) \ = \ \mathit{partByKey}\ k\ ((k,v):ts_2 + l_1) \\
 & \qquad\qquad\qquad\qquad\quad (ts_2,\ fs_2) \ = \ \mathit{partByKey}\ k\ l_2 \\
= & \quad \{\ \mathit{pieces} \text{ and } \mathit{flipApp}\ \} \\
 & (ts',\ \mathit{flipApp}(fs_1,fs_2)) \ \ \mathbf{where}\ (ts',\ fs_1) \ = \ \mathit{pieces}\ ((k,v):ts_2,l_1) \\
 & \qquad\qquad\qquad\qquad\qquad\quad (ts_2,\ fs_2) \ = \ \mathit{partByKey}\ k\ l_2
\end{array}
$$

where some obvious properties of *partByKey* are used. Condition (iii) for the other cases can also be checked in a similar way.

Heap sort. Gibbons (2007) presents a heap sort program as an example of metamorphism that does not satisfy the streaming condition. We demonstrate that the program meets the jigsaw condition. Although the heap sort algorithm cannot inherently be parallelized, we can implement it in the jigsaw model in which parallel evaluation is possible. This fact should not surprise us. The outcome is exactly a form of 'parallel bubble sort' as explained below. We basically follow Gibbons's definition of a heap sort except that the definition of the heap and its operating

https://doi.org/10.1017/S0956796812000391 Published online by Cambridge University Press

functions are abstracted out because two equivalent heaps may differ depending on their construction history.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Heap}\ a = \mathsf{Empty}\ \mid\ \langle\textit{nonempty heap whose elements have type } a\rangle \\
\textit{insert}\ :: (a, \mathsf{Heap}\ a) \to \mathsf{Heap}\ a \\
\textit{insert}\ (x,\ t)\ =\ \langle\textit{a heap obtained from the heap } t \textit{ by adding } x\rangle \\
\textit{splitMin}\ :: \mathsf{Heap}\ a \to \mathsf{Maybe}(a, \mathsf{Heap}\ a) \\
\textit{splitMin}\ \mathsf{Empty} \qquad\qquad = \mathsf{Nothing} \\
\textit{splitMin}\ t\ \mid\ t \neq \mathsf{Empty}\ =\ \mathsf{Just}(m, t') \\
\quad \textbf{where}\, m\ =\ \langle\textit{a minimum element in the heap } t\rangle \\
\qquad\quad\ t'\ =\ \langle\textit{a heap obtained by removing } m \textit{ from } t\rangle \\
\textit{heapSort}\ =\ \textit{unfoldr splitMin}\ \cdot\ \textit{foldr insert}\ \mathsf{Empty}
\end{array}
$$

where $\langle\textit{text}\rangle$ indicates an implementation abstracted by its specification. We require these functions only to satisfy a property

$$
\begin{array}{l}
\textit{splitMin}^{\#}(\textit{insert}(x,t)) = \\
\quad \textbf{if}\ x < m\ \textbf{then}\ (x,t)\ \textbf{else}\ (m, \textit{insert}(x,t'))\ \textbf{where}\ (m,t') = \textit{splitMin}^{\#}\ t
\end{array}
\tag{7}
$$

for any nonempty heap t, which promises that *splitMin* extracts a minimum element in a given heap. Jigsaw condition (i) holds from the definition of *splitMin*. Let us introduce ∞ for the (dummy) largest value, which is often used in sorting programs. We extend the definition of *insert* with $\textit{insert}\ (\infty, t) = t$ to deal with the ∞ value. Taking $h_0 = v_0 = \infty$ and

$$
\begin{array}{rl}
\textit{pieces}\ (x,y)\ \mid\ x < y & = (x,y) \\
\mid\ x \geqslant y & = (y,x),
\end{array}
$$

we obtain $\textit{heapSort} = \textit{jigsaw pieces}\ h_0\ v_0$. Jigsaw condition (ii) holds due to an extension of the definition of the *insert* function. For jigsaw condition (iii) with $v \neq \infty$ and a nonempty heap s,

$$
\begin{array}{cl}
 & \textit{splitMin}^{\#}(\textit{insert}(v,s)) \\
= & \quad \{\ \text{expected property (7)}\ \} \\
 & \textbf{if}\ v < m\ \textbf{then}\ (v,s)\ \textbf{else}\ (m, \textit{insert}(v,t))\ \textbf{where}\ (m,t) = \textit{splitMin}^{\#}\ s \\
= & \quad \{\ \textit{insert}\ \text{and}\ \textit{splitMin}\ \} \\
 & \textbf{if}\ v < m\ \textbf{then}\ (v, \textit{insert}(m,t))\ \textbf{else}\ (m, \textit{insert}(v,t))\ \textbf{where}\ (m,t) = \textit{splitMin}^{\#}\ s \\
= & \quad \{\ \textit{pieces}\ \} \\
 & (x, \textit{insert}(y,t))\ \textbf{where}\ (x,y) = \textit{pieces}(v,m)\ \textbf{and}\ (m,t) = \textit{splitMin}^{\#}\ s.
\end{array}
$$

Condition (iii) for other cases can similarly be checked. Since the *pieces* function swaps a wrong ordered pair of adjacent items, this exactly behaves as a bubble sort.

7 Concluding remarks

We have introduced a jigsaw model for metamorphisms. When it satisfies the jigsaw condition, a metamorphism can be implemented in our model such that there is room for parallel evaluation. It would be interesting to consider fusion and inversion on the jigsaw model, which we have left for future work.

https://doi.org/10.1017/S0956796812000391 Published online by Cambridge University Press

We do not claim that the jigsaw model will always provide an efficient implementation of metamorphisms. It is difficult to compare computational complexity between the jigsaw model and a naïve implementation of metamorphisms because their 'computation units' differ. For example, in m-to-n radix conversion, when the size of input is k and that of output is l, a naïve implementation of metamorphisms takes k-fold multiplication and l-fold division, while the jigsaw model takes kl-fold simple pattern matching. The most important feature of the jigsaw model is its flexibility in computation.

Acknowledgment

The author would like to thank Jeremy Gibbons and anonymous reviewers for their many insightful suggestions and corrections. He is also indebted to Hideya Iwasaki and Zhenjiang Hu for the helpful discussion they had with him.

References

Baader, F. & Nipkow, T. (1998) *Term Rewriting and All That*. Cambridge, UK: Cambridge University Press.

Bird, R. & Gibbons, J. (2003) Arithmetic coding with folds and unfolds. In *Proceedings of the 4th Advanced Functional Programming*, Lecture Notes in Computer Science, vol. 2638. Springer-Verlag, pp. 1–26.

Gibbons, J. (2007) Metamorphisms: Streaming representation-changers. *Sci. Comput. Program.* **65**(2), 108–139 (Elsevier).

Meijer, E., Fokkinga, M. & Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, vol. 523. Springer-Verlag, pp. 124–144.

https://doi.org/10.1017/S0956796812000391 Published online by Cambridge University Press