Programming Metamorphic Algorithms in Agda (Functional Pearl)*

HSIANG-SHANG KO, National Institute of Informatics, Japan

We conduct an experiment with interactive type-driven development in Agda, developing algorithms from their specifications encoded as intrinsic types, to see how useful the hints provided by Agda during an interactive development process can be. The algorithmic problem we choose is *metamorphisms*, whose definitional behaviour is consuming a data structure to compute an intermediate value and then producing a codata structure from that value, but there are other ways to compute metamorphisms. We develop Gibbons's [2007] streaming algorithm and Nakano's [2013] jigsaw model interactively with Agda, turning intuitive ideas about these algorithms into formal conditions and programs that are correct by construction.

CCS Concepts: • **Software and its engineering → Functional languages**;

Additional Key Words and Phrases: Dependently Typed Programming, Interactive Type-Driven Development, Agda, Metamorphisms

ACM Reference Format:
Hsiang-Shang Ko. 2018. Programming Metamorphic Algorithms in Agda (Functional Pearl). *Proc. ACM Program. Lang.* 0, 0, Article 0 (March 2018), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTERACTIVE TYPE-DRIVEN DEVELOPMENT

Why do we want to program with full dependent types? For larger-scale proofs, writing proof terms in a dependently typed language is usually much less efficient than working with a proof assistant with decent automation. What has been more successful with dependently typed programming (DTP) — in particular with the use of indexed datatypes — is intrinsic hygienic guarantee, which tends to work better for smaller, hand-crafted programs: if we need to, for example, track the bounds of indices or work with well-typed syntax trees, let us encode bound and typing constraints in the datatypes of indices and syntax trees so that the type system can help the programmer to enforce the constraints. But this kind of guarantee per se is also achievable extrinsically with proof assistants — we know how to prove theorems saying that indices are always within bounds or that a program transformation always produces well-typed programs. The ability to offer this kind of guarantee is not what makes DTP truly shine. Rather, the most distinguishing and fascinating aspect of DTP is that it makes the paradigm of *interactive type-driven development* much more powerful: the more properties we encode intrinsically into types, the more hints the type-checker can offer the programmer during an interactive development process, so that the more heavyweight typing becomes an aid rather than a burden. A typical example is making the terms of an embedded domain-specific language (DSL) intrinsically typed, so that we are always aware of the precise types of the terms (in the DSL's type system) and guided to construct only well-typed terms.

A natural follow-up question is: how well does interactive type-driven development scale? For example, if we encode a complete specification of an algorithmic problem in a type, will the hints provided by the type-checker be useful enough to guide us to an algorithm? Thankfully, we now have dependently typed languages like Agda [Norell 2007] and Idris [Brady 2013] whose

*Draft manuscript (17th March 2018)

Author's address: Hsiang-Shang Ko, Information Systems Architecture Science Research Division, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430, Japan, hsiang-shang@nii.ac.jp.

2018. 2475-1421/2018/3-ART0 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

interactive development environment is mature enough for doing some experiments — that is, trying to program some algorithms interactively with the type-checker. In this paper we will use Agda 2.5.2 with Standard Library version 0.13.[2] We also need an algorithmic problem, and we had better start with one that is not too complicated and also not too trivial. One such problem that comes to mind is metamorphisms.

2 METAMORPHISMS

A *metamorphism* [Gibbons 2007] consumes a data structure to compute an intermediate value and then produces a new data structure using the intermediate value as the seed. Following Gibbons [2007], we will focus on *list metamorphisms*, i.e., metamorphisms consuming and producing lists. Two of the examples of list metamorphisms given by Gibbons are:

- *Base conversion for fractions*: A list of digits representing a fractional number in a particular base (e.g., 0.625_{10}) can be converted to another list of digits in a different base (e.g., 0.101_2). The conversion is a metamorphism because we can consume an input list of digits to compute the value represented by the list, and then produce an output list of digits representing the same value. Note that even when the input list is finite, the output list may have to be infinite — $0.1_3 = 0.333\ldots_{10}$, for example.
- *Heapsort*: An input list of numbers is consumed by pushing every element into a min-heap (priority queue), from which we then pop all the elements, from the smallest to the largest, producing a sorted output list.

Metamorphisms are an interesting subject because they can be computed by a number of different algorithms [Gibbons 2007; Nakano 2013]. Knowing the existence of these algorithms, our experiment will have a clearer direction to follow and be more likely to succeed. However, rather than merely implementing them in Agda, we should try to reinvent as much of the algorithms as possible, so that we can feel what it is like to get from a specification to an algorithm with the help of Agda, and perhaps gain a different and/or better understanding of these algorithms.

Let us first say more precisely what metamorphisms are. Formally, a metamorphism is a *fold* followed by an *unfold*, the former consuming a finite data structure and the latter producing a potentially infinite codata structure.

▶ *Lists for consumption.* For list metamorphisms, the inputs to be consumed are the standard finite lists:[3]

```
data List (A : Set) : Set where
  []  : List A
  _::_ : A → List A → List A
```

The *foldr* operator subsumes the elements (of type A) of a list into a state (of type S) using a "right algebra" $(\triangleleft) : A \to S \to S$ and an initial/empty state $e : S$:[4]

```
foldr : {A S : Set} → (A → S → S) → S → List A → S
foldr (◁) e []       = e
foldr (◁) e (a :: as) = a ◁ foldr f e as
```

[2] The code in this paper is collected in the supplementary Agda source file.

[3] In Agda, a name with underscores like _::_ can be used as an operator, and the underscores indicate where the arguments go. As an exception, in this paper we often write a binary infix operator like $\_\triangleleft\_$ in Haskell syntax like $(\triangleleft)$, which deviates from Agda syntax.

[4] In the type of a function, arguments wrapped in curly brackets are implicit, and can be left out (if they are inferable by Agda) when applying the function.

With *foldr*, a list is consumed from the right. Dually, the *foldl* operator consumes a list from the left using a "left algebra" $(\triangleright) : S \to A \to S$:

```
foldl : {A S : Set} → (S → A → S) → S → List A → S
foldl (▷) e []       = e
foldl (▷) e (a :: as) = foldl (▷) (e ▷ a) as
```

A list metamorphism can use either *foldr* or *foldl* in its consuming part, and we will see both kinds in the paper. We will refer to a list metamorphism using *foldr* as a "right metamorphism", and one using *foldl* as a "left metamorphism".

► *Colists for production.* For the producing part of list metamorphisms, where we need to produce potentially infinite lists, in a total language like Agda we can no longer use List, whose elements are necessarily finite; instead, we should switch to a *codatatype* of *colists*, which are potentially infinite. Dual to a datatype, which is defined by all the possible ways to *construct* its elements, a codatatype is defined by all the possible ways to *deconstruct* (or observe) its elements. For a colist, we only need one way of deconstruction: exposing the colist's outermost structure, which is either empty or a pair of a head element and a tail colist. In Agda this can be expressed as a coinductive record type CoList with only one field, which is a sum (empty or non-empty) structure; for cosmetic reasons, this sum structure is defined (mutually recursively with CoList) as a datatype CoListF:[5]

```
mutual
  record CoList (B : Set) : Set where
    coinductive
    field
      decon : CoListF B
  data CoListF (B : Set) : Set where
    []  : CoListF B
    _::_ : B → CoList B → CoListF B
```

Note that decon denotes a function of type $\{B : \mathsf{Set}\} \to \mathsf{CoList}\ B \to \mathsf{CoListF}\ B$, and plays the role of the deconstructor of CoList. Now we can define the standard *unfoldr* operator, which uses a coalgebra $g : S \to \mathsf{Maybe}\ (B \times S)$ to unfold a colist from a given state:[6]

```
unfoldr : {B S : Set} → (S → Maybe (B × S)) → S → CoList B
decon (unfoldr g s) with g s
decon (unfoldr g s) | nothing     = []
decon (unfoldr g s) | just (b , s') = b :: unfoldr g s'
```

The operator is defined with copattern matching [Abel et al. 2013]: To define *unfoldr g s*, which is a colist, we need to specify what will result if we deconstruct it, i.e., what decon (*unfoldr g s*) will compute to. This depends on whether *g* can produce anything from *s*, so, using the **with** construct, we introduce *g s* as an additional "argument", on which we can then perform a pattern match. If *g s* is nothing, then the resulting colist will be empty — that is, decon (*unfoldr g s*) will compute to []; otherwise, *g s* is just (b, s') for some *b* and s', and the resulting colist will have *b* as its head and *unfoldr g* s' as its tail — that is, decon (*unfoldr g s*) will compute to b :: *unfoldr g* s'.

To be more concrete, let us describe our two examples — base conversion for fractions and heapsort — explicitly as metamorphisms.

[5] Agda supports constructor overloading, so we are allowed to reuse [] and _::_ as the names of the constructors of CoListF.
[6] The datatype Maybe *X* has two constructors, nothing : Maybe *X* and just : $X \to$ Maybe *X*.

▶ *Base conversion for fractions.* Suppose that the input and output bases are $b_i : \mathbb{N}$ and $b_o : \mathbb{N}$ — in $0.625_{10} = 0.101_2$, for example, $b_i = 10$ and $b_o = 2$. We represent fractions as (co)lists of digits (of type $\mathbb{N}$) starting from the most significant digit — for example, 0.625 is represented as $6 :: 2 :: 5 :: []$. To make the story short later,[7] we describe base conversion for fractions as a left metamorphism:

$$\mathit{unfoldr}\ g_{\mathrm{C}} \circ \mathit{foldl}\ \_\triangleright_{\mathrm{C}}\_\ e_{\mathrm{C}}$$

where the state type is $S_{\mathrm{C}} = \mathbb{Q} \times \mathbb{Q} \times \mathbb{Q}$, which are triples of rationals of the form $(v\,,\ w_i\,,\ w_o)$ where v is an accumulator, w_i the weight of the incoming input digit, and w_o the weight of the outgoing output digit. The initial state e_{C} is $(0\,,\ {}^1\!/_{b_i}\,,\ {}^1\!/_{b_o})$. The left algebra $(\triangleright_{\mathrm{C}})$ adds the product of the current input digit and its weight to the accumulator, and updates the input weight in preparation for the next input digit:

$$\begin{aligned}&(\triangleright_{\mathrm{C}}) : S_{\mathrm{C}} \to \mathbb{N} \to S_{\mathrm{C}}\\ &(v\,,\ w_i\,,\ w_o) \triangleright_{\mathrm{C}} d\ =\ (v + d \times w_i\,,\ {}^{w_i}\!/_{b_i}\,,\ w_o)\end{aligned}$$

while the coalgebra g_{C} produces an output digit and updates the accumulator and the next output weight if the accumulator is not yet zero:

$$\begin{aligned}g_{\mathrm{C}}\ (v\,,\ w_i\,,\ w_o)\ =\ &\mathbf{if}\ v > 0\ \mathbf{then}\ \mathbf{let}\ d = \lfloor {}^v\!/_{w_o} \rfloor; r = v - d \times w_o\\ &\qquad\qquad\qquad\qquad \mathbf{in}\ \mathsf{just}\ (d\,,\ (r\,,\ w_i\,,\ {}^{w_o}\!/_{b_o}))\\ &\qquad\quad \mathbf{else}\ \ \mathsf{nothing}\end{aligned}$$

For the example $0.625_{10} = 0.101_2$, the metamorphism first consumes the input digits using $(\triangleright_{\mathrm{C}})$:

$$(0\,,\ 0.1\,,\ 0.5) \overset{6}{\mapsto} (0.6\,,\ 0.01\,,\ 0.5) \overset{2}{\mapsto} (0.62\,,\ 0.001\,,\ 0.5) \overset{5}{\mapsto} (0.625\,,\ 0.0001\,,\ 0.5)$$

and then produces the output digits using g_{C}:

$$\begin{aligned}(0.625\,,\ 0.0001\,,\ 0.5) &\overset{1}{\mapsto} (0.125\,,\ 0.0001\,,\ 0.25) \overset{0}{\mapsto} (0.125\,,\ 0.0001\,,\ 0.125)\\ &\overset{1}{\mapsto} (0\,,\ 0.0001\,,\ 0.0625) \not\mapsto\end{aligned}$$

▶ *Heapsort.* Let Heap be an abstract type of min-heaps on some totally ordered set Val, equipped with three operations:

$$\begin{aligned}&\mathit{empty}\ \ : \mathsf{Heap}\\ &\mathit{push}\ \ \ : \mathsf{Val} \to \mathsf{Heap} \to \mathsf{Heap}\\ &\mathit{popMin} : \mathsf{Heap} \to \mathsf{Maybe}\ (\mathsf{Val} \times \mathsf{Heap})\end{aligned}$$

where *empty* is the empty heap, *push* adds an element into a heap, and *popMin* returns a minimum element and the rest of the input heap if and only if the input heap is non-empty. Then heapsort can be directly described as a right metamorphism:

$$\mathit{unfoldr}\ \mathit{popMin} \circ \mathit{foldr}\ \mathit{push}\ \mathit{empty}$$

[7] Gibbons [2007, Section 4.2] gives a more complete story, where base conversion for fractions is first described as a right metamorphism with simple states (consisting of only an accumulator), and then transformed to a left metamorphism with more complex states.

3 SPECIFICATION OF METAMORPHISMS IN TYPES

In the rest of this paper we will develop (actually, reinvent) several *metamorphic algorithms*, which compute a metamorphism but do not take the form of a fold followed by an unfold. Rather than proving extrinsically that these algorithms satisfy their metamorphic specifications, we will encode metamorphic specifications intrinsically in types, such that any type-checked program is a correct metamorphic algorithm.

▶ *Algebraic ornamentation.* The encoding is based on McBride's [2011] *algebraic ornamentation*. Given a right algebra $(\triangleleft) : A \to S \to S$ and $e : S$, we can partition List A into a family of types AlgList A $(\triangleleft)$ $e : S \to$ Set indexed by S such that (conceptually) every list as falls into the type AlgList A $(\triangleleft)$ e (*foldr* $(\triangleleft)$ e as). The definition of AlgList is obtained by "fusing" *foldr* into List:

```
data AlgList (A {S} : Set) ((◁) : A → S → S) (e : S) : S → Set where
  []    : AlgList A (◁) e e
  _::_ : (a : A) → {s : S} → AlgList A (◁) e s → AlgList A (◁) e (a ◁ s)
```

The empty list is classified under the index e = *foldr* $(\triangleleft)$ e []. For the cons case, if a tail as is classified under s, meaning that *foldr* $(\triangleleft)$ e $as = s$, then the whole list $a :: as$ should be classified under $a \triangleleft s$ since *foldr* $(\triangleleft)$ e $(a :: as) = a \triangleleft$ *foldr* $(\triangleleft)$ e $as = a \triangleleft s$.

▶ *Coalgebraic ornamentation.* Dually, given a coalgebra $g : S \to$ Maybe $(B \times S)$, we can refine CoList B into a family of types CoalgList B $g : S \to$ Set such that a colist falls into CoalgList B g s if it is unfolded from s using g. (Note that, extensionally, every CoalgList B g s has exactly one inhabitant; intensionally there may be different ways to describe/compute that inhabitant, though.) Again the definition of CoalgList is obtained by fusing *unfoldr* into CoList:[8]

```
mutual
  record CoalgList (B {S} : Set) (g : S → Maybe (B × S)) (s : S) : Set where
    coinductive
    field
      decon : CoalgListF B g s
  data CoalgListF (B {S} : Set) (g : S → Maybe (B × S)) : S → Set where
    ⟨_⟩      : {s : S} → g s ≡ nothing → CoalgListF B g s
    _::⟨_⟩_ : (b : B) → {s s' : S} → g s ≡ just (b , s') → CoalgList B g s' → CoalgListF B g s
```

Deconstructing a colist of type CoalgList B g s can lead to two possible outcomes: the colist can be empty, in which case we also get an equality proof that g s is nothing, or it can be non-empty, in which case we know that g s produces the head element, and that the tail colist is unfolded from the next state s' produced by g s.

Let A, B, S : Set throughout the rest of this paper[9] — we will assume that A is the type of input elements, B the type of output elements, and S the type of states. We will also consistently let $(\triangleleft) : A \to S \to S$ denote a right algebra, $(\triangleright) : S \to A \to S$ a left algebra, $e : S$ an initial/empty state, and $g : S \to$ Maybe $(B \times S)$ a coalgebra.

▶ *Right metamorphisms.* Any program of type:

$$\{s : S\} \to \text{AlgList } A\ (\triangleleft)\ e\ s \to \text{CoalgList } B\ g\ s$$

implements the right metamorphism *unfoldr* $g \circ$ *foldr* $(\triangleleft)$ e, since the indexing enforces that the input list folds to s, from which the output colist is then unfolded.

[8] The operator $\_\equiv\_ : \{A : \text{Set}\} \to A \to A \to \text{Set}$ is the equality type constructor in the Agda Standard Library. An equality type $x \equiv y$ is inhabited by the refl constructor if x has the same normal form as y, or uninhabited otherwise.
[9] That is, think of the code in the rest of this paper as contained in a module with parameters A, B, S : Set.

▶ *Left metamorphisms.* What about left metamorphisms? Conveniently, we do not need to define another variant of AlgList due to an old trick that expresses *foldl* in terms of *foldr*. Given a list $as : \mathsf{List}\ A$, think of the work of *foldl* $(\triangleright)\ e\ as$ as (i) partially applying *flip* $(\triangleright) : A \to S \to S$ (where *flip* $f\ x\ y = f\ y\ x$) to every element of *as* to obtain state transformations of type $S \to S$, (ii) composing the state transformations from left to right, and finally (iii) applying the resulting composite transformation to *e*. The left-to-right order appears only in step (ii), which, in fact, can also be performed from right to left since function composition is associative. Formally, we have:

$$\mathit{foldl}\ (\triangleright)\ e\ as\ =\ \mathit{foldr}\ (\mathit{from\text{-}left\text{-}alg}\ (\triangleright))\ \mathit{id}\ as\ e$$

where

$$\mathit{from\text{-}left\text{-}alg} : \{A\ S : \mathsf{Set}\} \to (S \to A \to S) \to A \to (S \to S) \to (S \to S)$$
$$\mathit{from\text{-}left\text{-}alg}\ (\triangleright)\ a\ t\ =\ t \circ \mathit{flip}\ (\triangleright)\ a$$

and $\mathit{id} : \{A : \mathsf{Set}\} \to A \to A$ is the identity function. The type of left metamorphic algorithms can then be specified as:

$$(s : S) \to \{h : S \to S\} \to \mathsf{AlgList}\ A\ (\mathit{from\text{-}left\text{-}alg}\ (\triangleright))\ \mathit{id}\ h \to \mathsf{CoalgList}\ B\ g\ (h\ s)$$

which says that if the initial state is *s* and the input list folds to a state transformation *h*, then the output colist should be unfolded from $h\ s$.

4 METAMORPHISMS, DEFINITIONALLY

To warm up, let us start from the left metamorphic type and implement the most straightforward algorithm that strictly follows the definition of metamorphisms, **c**onsuming all inputs **b**efore **p**roducing outputs:

```
cbp : (s : S) → {h : S → S} → AlgList A (from-left-alg (▷)) id h → CoalgList B g (h s)
cbp s as {AlgList A (from-left-alg (▷)) id h} = {CoalgList B g (h s)}₀
```

We will try to recreate in this paper what it feels like to program interactively with Agda.[10] During interaction, we can leave "holes" in programs and fill or refine them, often with Agda's help. Such a hole is called an *interaction point* or a *goal*, of which the green-shaded part above is an example. At goals, Agda can be instructed to provide various information and even perform some program synthesis. One most important piece of information for a goal is its expected type, which we always display in curly brackets. Goals are numbered so they can be referred to in the text. At goals, we can also query the types of the variables in scope; whenever the type of a variable needs to be displayed, we will annotate the variable with its type in yellow-shaded subscript (which is not part of the program text). In the program above, we annotate *as* with its type because the expected type at Goal 0 refers to *h*, which is the index in the type of *as*.

Now let us try to develop the program. We are trying to consume the input list first, so we pattern match on the argument *as* to see if there is anything to consume. In Agda this is as easy as putting *as* into Goal 0 and firing a "case split" command (`C-c C-c`); the program will then be split into two clauses, listing all possible cases of *as*:

```
cbp : (s : S) → {h : S → S} → AlgList A (from-left-alg (▷)) id h → CoalgList B g (h s)
cbp s []       = {CoalgList B g s}₁
cbp s (a :: as {AlgList A (from-left-alg (▷)) id h}) = {CoalgList B g (h (s ▷ a))}₂
```

[10] More precisely, it is the interactive development environment included in the Agda package that we interact with. This interactive development environment is implemented as an Emacs mode, which explains the form of the commands that we will see below.

Now Goal 0 is gone, and two new goals appear. Note that the expected types of the two new goals have changed: at Goal 1, for example, we see that the output colist should be unfolded directly from the initial state s since the input list is empty. By providing sufficient type information, Agda can keep track of such relationships for us! We continue to interact with and refine these two new goals.

▶ *Consumption.* If there is something to consume, that is, the input list is non-empty, we go into Goal 2, where we keep consuming the tail *as* but from a new state:

```
cbp : (s : S) → {h : S → S} → AlgList A (from-left-alg (▷)) id h → CoalgList B g (h s)
cbp s []       = {CoalgList B g s}₁
cbp s (a :: as) = cbp {S}₃ as
```

What is this new state? It should be the one obtained by subsuming a into s, i.e., $s \triangleright a$. Agda knows this too, in fact — firing the "Auto" command (`C-c C-a`) at Goal 3 completely fulfills it:

```
cbp : (s : S) → {h : S → S} → AlgList A (from-left-alg (▷)) id h → CoalgList B g (h s)
cbp s []       = {CoalgList B g s}₁
cbp s (a :: as) = cbp (s ▷ a) as
```

▶ *Production.* If there is nothing more to consume, that is, the input list is empty, we go into Goal 1, where we should produce the output colist; to specify the colist, we should say what will result if we deconstruct the colist. That is, we perform a copattern match (which can be done by Agda if we give it the case splitting command (`C-c C-c`) without specifying a variable):

```
cbp : (s : S) → {h : S → S} → AlgList A (from-left-alg (▷)) id h → CoalgList B g (h s)
decon (cbp s []) = {CoalgListF B g s}₄
cbp s (a :: as)  = cbp (s ▷ a) as
```

The result of deconstruction depends on whether g can produce anything from the current state s, so we pattern match on $g\ s$, splitting Goal 4 into Goals 5 and 6:

```
cbp : (s : S) → {h : S → S} → AlgList A (from-left-alg (▷)) id h → CoalgList B g (h s)
decon (cbp s []) with g s
decon (cbp s []) | nothing     = {CoalgListF B g s}₅
decon (cbp s []) | just (b , s') = {CoalgListF B g s}₆
cbp s (a :: as)  = cbp (s ▷ a) as
```

If $g\ s$ is nothing (Goal 5), the output colist is empty; otherwise $g\ s$ is just (b, s') for some b and s' (Goal 6), in which case we use b as the head and go on to produce the tail from s'. We therefore refine the two goals manually into:

```
cbp : (s : S) → {h : S → S} → AlgList A (from-left-alg (▷)) id h → CoalgList B g (h s)
decon (cbp s []) with g s
decon (cbp s []) | nothing     = ⟨ {g s ≡ nothing}₇ ⟩
decon (cbp s []) | just (b , s') = b ::⟨ {g s ≡ just (b , s')}₈ ⟩ cbp s' []
cbp s (a :: as)  = cbp (s ▷ a) as
```

(More specifically, the refinement from Goal 5 to Goal 7, for example, can be done by filling "⟨ ? ⟩" into Goal 5 and give the "refine" command (`C-c C-r`) to Agda.)

We are now required to discharge equality proof obligations about $g\ s$, and the obligations exactly correspond to the results of the **with**-matching. This is precisely a situation in which the *inspect* idiom in the Agda Standard Library can help. With *inspect*, we can obtain an equality proof of the right type in each of the cases of the **with**-matching:

```
module ConsumingBeforeProducing
  ((▷) : S → A → S) (g : S → Maybe (B × S))
  where
  cbp : (s : S) → {h : S → S} → AlgList A (from-left-alg (▷)) id h → CoalgList B g (h s)
  decon (cbp s []) with g s        | inspect g s
  decon (cbp s []) | nothing       | [eq] = ⟨ eq ⟩
  decon (cbp s []) | just (b , s') | [eq] = b ::⟨ eq ⟩ cbp s' []
  cbp s (a :: as) = cbp (s ▷ a) as
```

Fig. 1. Definitional implementation of metamorphisms

```
cbp : (s : S) → {h : S → S} → AlgList A (from-left-alg (▷)) id h → CoalgList B g (h s)
decon (cbp s []) with g s        | inspect g s
decon (cbp s []) | nothing       | [eq g s ≡ nothing    ] = ⟨ {g s ≡ nothing}₇ ⟩
decon (cbp s []) | just (b , s') | [eq g s ≡ just (b , s')] = b ::⟨ {g s ≡ just (b , s')}₈ ⟩ cbp s' []
cbp s (a :: as) = cbp (s ▷ a) as
```

Both goals can now be discharged with *eq*, and we arrive at a complete program, shown in Figure 1. As explained in Section 3, the correctness of this program is established by type-checking — no additional proofs are needed.

5 THE STREAMING ALGORITHM

As Gibbons [2007] noted, metamorphisms in general cannot be automatically optimised in terms of time and space, but in some cases it is possible to compute a list metamorphism using a *streaming algorithm*, which can produce an initial segment of the output colist from an initial segment of the input list. For example, when converting 0.625_{10} to 0.101_2, after consuming the first decimal digit 6 and reaching the state $(0.6\,,\ 0.01\,,\ 0.5)$, we can directly produce the first binary digit 1 because we know that the number will definitely be greater than 0.5. Streaming is not always possible, of course. Heapsort is a counterexample: no matter how many input elements have been consumed, it is always possible that a minimum element — which should be the first output element — has yet to appear, and thus we can never produce the first output element before we see the whole input list. There should be some condition under which we can stream metamorphisms, and we should be able to discover such condition if we program a streaming algorithm together with Agda, which knows what metamorphisms are and can provide us with semantic hints regarding what conditions need to be introduced to make the program a valid metamorphic algorithm.

We start from the same left metamorphic type:

```
stream : (s : S) → {h : S → S} → AlgList A (from-left-alg (▷)) id h → CoalgList B g (h s)
stream s as AlgList A (from-left-alg (▷)) id h = {CoalgList B g (h s)}₀
```

In contrast to *cbp* (Section 4/Figure 1), this time we try to produce using g whenever possible, so our first step is to pattern match on $g\ s$ (and we also introduce decon and *inspect*, which will be needed like in *cbp*):

```
stream : (s : S) → {h : S → S} → AlgList A (from-left-alg (▷)) id h → CoalgList B g (h s)
decon (stream s as AlgList A (from-left-alg (▷)) id h) with g s | inspect g s
decon (stream s as) | nothing       | [eq] = {CoalgListF B g (h s)}₁
decon (stream s as) | just (b , s') | [eq] = {CoalgListF B g (h s)}₂
```

▶ *Consumption.* For Goal 1, we cannot produce anything since $g\ s$ is nothing, but this does not mean that the output colist is empty — we may be able to produce something once we consume the input list and advance to a new state. We therefore pattern match on the input list, splitting Goal 1 into Goals 3 and 4:

```
stream : (s : S) → {h : S → S} → AlgList A (from-left-alg (▷)) id h → CoalgList B g (h s)
decon (stream s as) with g s        | inspect g s
decon (stream s []) | nothing       | [eq] = {CoalgListF B g s}₃
decon (stream s (a :: as AlgList A (from-left-alg (▷)) id h ))
                    | nothing       | [eq] = {CoalgListF B g (h (s ▷ a))}₄
decon (stream s as) | just (b , s') | [eq] = {CoalgListF B g (h s)}₂
```

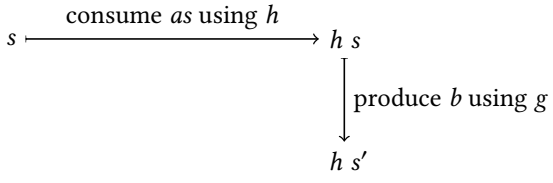
The two goals are similar to what we have seen in *cbp*. At Goal 3, there is nothing more in the input list to consume, so we should end production and emit an empty colist, while for Goal 4 we should advance to the new state $s \triangleright a$ and set the tail as as the list to be consumed next:

```
stream : (s : S) → {h : S → S} → AlgList A (from-left-alg (▷)) id h → CoalgList B g (h s)
decon (stream s as      ) with g s        | inspect g s
decon (stream s []      ) | nothing       | [eq] = ⟨ eq ⟩
decon (stream s (a :: as)) | nothing      | [eq] = decon (stream (s ▷ a) as)
decon (stream s as      ) | just (b , s') | [eq] = {CoalgListF B g (h s)}₂
```

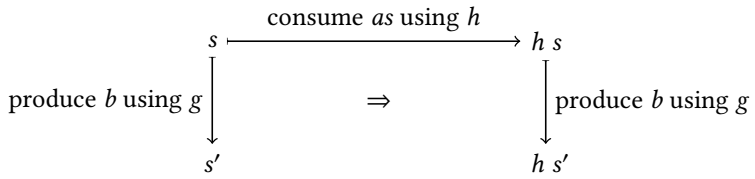
▶ *Production.* Goal 2 is the interesting case. Using g, from the current state s we can produce b, which we set as the head of the output colist, and advance to a new state s', from which we produce the tail of the colist:

```
stream : (s : S) → {h : S → S} → AlgList A (from-left-alg (▷)) id h → CoalgList B g (h s)
decon (stream s as      ) with g s        | inspect g s
decon (stream s []      ) | nothing       | [eq] = ⟨ eq ⟩
decon (stream s (a :: as)) | nothing      | [eq] = decon (stream (s ▷ a) as)
decon (stream s as AlgList A (from-left-alg f) id h )
                           | just (b , s') | [eq g s ≡ just (b , s') ] =
                                b ::⟨ {g (h s) ≡ just (b , h s')}₅ ⟩ stream s' as
```

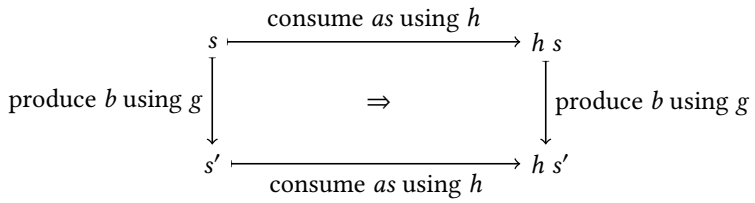
▶ *The streaming condition.* Now Agda gives us a non-trivial proof obligation at Goal 5 — what does it mean? The left-hand side $g\ (h\ s)$ is trying to produce using g from the state $h\ s$, where h is the state transformation resulting from consuming the entire input list as (since h is the index in the type of as), and the whole equality says that this has to produce a specific result. Drawing this as a state transition diagram:



We already have in the context a similar-looking equality, namely $eq : g\ s \equiv \text{just}\ (b\,,\ s')$, which we can superimpose on the diagram:



We also put in an implication arrow to indicate more explicitly that $g\ s \equiv \text{just}\ (b\,,\ s')$ is a premise, from which we should derive $g\ (h\ s) \equiv \text{just}\ (b\,,\ h\ s')$. Now it is tempting, and indeed easy, to complete the diagram:



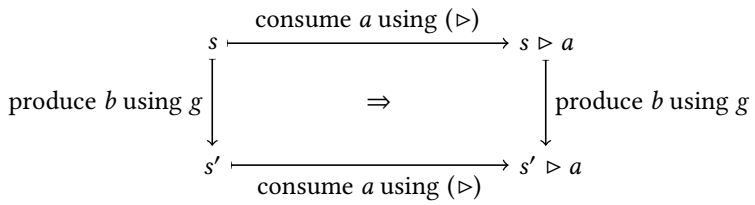
(1)

This is a kind of commutativity of production and consumption: From the initial state s, we can either

- apply g to s to produce b and reach a new state s', and then apply h to consume the list and update the state to $h\ s'$, or
- apply h to s to consume the list and update the state to $h\ s$, and then apply g to $h\ s$ to produce an element and reach a new state.

If the first route is possible, the second route should also be possible, and the outcomes should be the same — doing production using g and consumption using h in whichever order should emit the same element and reach the same final state. This will not be true in general, and should be formulated as a condition of the streaming algorithm.

But the above commutativity (1) of g and h — which is commutativity of one step of production (using g) and multiple steps of consumption (of the entire input list, using h) — may not be a good condition to impose. If we require instead that g and $(\triangleright)$ commute, this commutativity of single-step production and consumption will be easier for the algorithm user to verify:



(2)

This is Gibbons's [2007] *streaming condition*. In our development of *stream*, we need to assume that a proof of the streaming condition is available:

```
constant streaming-condition : {a : A} {b : B} {s s′ : S} →
                               g s ≡ just (b , s′) → g (s ▷ a) ≡ just (b , s′ ▷ a)
```

We use a hypothetical **constant** keyword here to emphasise that *streaming-condition* is a constant made available to us and does not need to be defined. In the complete program in Figure 2, the

functions defined in this section are collected in a module, and *streaming-condition* is made a parameter of this module.

▶ *Wrapping up.* Back to Goal 5, where we should prove the commutativity of g and h. All it should take is a straightforward induction to extend the streaming condition along the axis of consumption — so straightforward, in fact, that Agda can do most of the work for us! We know that we need a helper function *streaming-lemma* that performs induction on *as* and uses *eq* as a premise; by filling *streaming-lemma as eq* into Goal 5 and firing a "helper type" command (`C-c C-h`), Agda can generate a type for *streaming-lemma*, which, after removing some over-generalisations and unnecessary definition expansions, is:

```
streaming-lemma : {b : B} {s s′ : S} {h : S → S} → AlgList A (from-left-alg (▷)) id h →
                  g s ≡ just (b , s′) → g (h s) ≡ just (b , h s′)
streaming-lemma as AlgList A (from-left-alg (▷)) id h eq g s ≡ just (b , s′) = {g (h s) ≡ just (b , h s′)}₆
```

Agda then accepts *streaming-lemma as eq* as a type-correct term for Goal 5, completing the definition of *stream*.

Now all that is left is the body of *streaming-lemma* (Goal 6). If we give a hint that case splitting is needed (`-c`), Auto can complete the definition of *streaming-lemma* on its own, yielding (modulo one cosmetic variable renaming):

```
streaming-lemma : {b : B} {s s′ : S} {h : S → S} → AlgList A (from-left-alg (▷)) id h →
                  g s ≡ just (b , s′) → g (h s) ≡ just (b , h s′)
streaming-lemma []       eq = eq
streaming-lemma (a :: as) eq = streaming-lemma as (streaming-condition eq)
```

The complete program is shown in Figure 2.

▶ *Streaming base conversion for fractions.* We have (re)discovered the streaming condition, but does it hold for the base conversion metamorphism $\mathit{unfoldr}\ g_C \circ \mathit{foldl}\ (\triangleright_C)\ e_C$ given in Section 2? Actually, no. The problem is that g_C can be too eager to produce an output digit. In $0.625_{10} = 0.101_2$, for example, after consuming the first decimal digit 6, we can safely use g_C to produce the first two binary digits 1 and 0, reaching the state $(0.1\,,\ 0.01\,,\ 0.125)$. From this state, g_C will produce a third binary digit 0, but this can be wrong if there are more input digits to consume — indeed, in our example the next input digit is 5, and the accumulator will go up to $0.1 + 5 \times 0.01 = 0.15$, exceeding the next output weight 0.125, and hence the next output digit should be 1 instead. To allow streaming, we should make g_C more conservative, producing an output digit only when the accumulator will not go up too much to change the produced output digit whatever the unconsumed input digits might be. We therefore revise g_C to check an extra condition (underlined below) before producing:

$$
\begin{aligned}
g'_C\ (v\,,\ w_i\,,\ w_o) \ =\ & \textbf{let}\ d = \lfloor v/w_o \rfloor; r = v - d \times w_o \\
& \textbf{in}\ \ \textbf{if}\ v > 0 \wedge \underline{r + b_i \times w_i \leqslant w_o}\ \textbf{then}\ \mathsf{just}\ (d\,,\ (r\,,\ w_i\,,\ {}^{w_o}/_{b_o})) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \ \mathsf{nothing}
\end{aligned}
$$

In this extra condition, r is the updated accumulator after producing an output digit, and $b_i \times w_i$ is the supremum value attainable by the unconsumed input digits. If the sum $r + b_i \times w_i$ exceeds w_o, the output digit may have to be increased, in which case we should not produce the digit just yet. After this revision, the streaming condition holds for g'_C and $(\triangleright_C)$.

Once all the input digits have been consumed, however, g'_C can be too conservative and does not produce output digits even when the accumulator is not zero. This is another story though — the interested reader is referred to Gibbons [2007, Section 4.4].

```
module Streaming
  ((▷) : S → A → S) (g : S → Maybe (B × S))
  (streaming-condition : {a : A} {b : B} {s s′ : S} →
                         g s ≡ just (b , s′) → g (s ▷ a) ≡ just (b , s′ ▷ a))
  where
  streaming-lemma : {b : B} {s s′ : S} {h : S → S} → AlgList A (from-left-alg (▷)) id h →
                    g s ≡ just (b , s′) → g (h s) ≡ just (b , h s′)
  streaming-lemma []       eq = eq
  streaming-lemma (a :: as) eq = streaming-lemma as (streaming-condition eq)
  stream : (s : S) → {h : S → S} → AlgList A (from-left-alg (▷)) id h → CoalgList B g (h s)
  decon (stream s as      ) with g s       | inspect g s
  decon (stream s []      ) | nothing      | [ eq ] = ⟨ eq ⟩
  decon (stream s (a :: as)) | nothing     | [ eq ] = decon (stream (s ▷ a) as)
  decon (stream s as      ) | just (b , s′) | [ eq ] = b ::⟨ streaming-lemma as eq ⟩ stream s′ as
```

Fig. 2. The streaming algorithm

6 THE JIGSAW MODEL

Let us now turn to right metamorphisms. Recall that a right metamorphic type has the form:

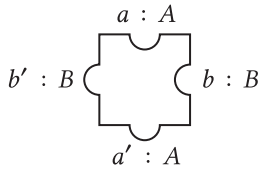
$$\{s : S\} \to \mathsf{AlgList}\ A\ (\triangleleft)\ e\ s \to \mathsf{CoalgList}\ B\ g\ s$$

which, unlike a left metamorphic type, does not have an initial state as one of its arguments. The only state appearing in the type is the implicit argument $s : S$, which is the intermediate state reached after consuming the entire input list, and it is unreasonble to assume that this intermediate state is also given at the start of a metamorphic computation — for example, when computing the heapsort metamorphism, we would not expect a heap containing all the elements of the input list to be given to an algorithm as input. This suggests that s plays a role only in the type-level specification, and we should avoid using s in the actual computation — that is, s should be computationally irrelevant, and could be somehow erased. Correspondingly, the indices and proofs in AlgList and CoalgList could all be erased eventually, turning a program with a right metamorphic type into one that maps plain lists to colists. Does this mean that we can bypass computation with states and just work with list elements to compute a metamorphism? Surprisingly, Nakano [2013] has such a computation model, in which it is possible to compute a metamorphism without using the states mentioned in its specification! (By contrast, in *cbp* (Section 4/Figure 1) and *stream* (Section 5/Figure 2), we can hope to erase the indices and proofs in AlgList and CoalgList but not the input state, which is used relevantly in the computation.)

6.1 Computation in the Jigsaw Model

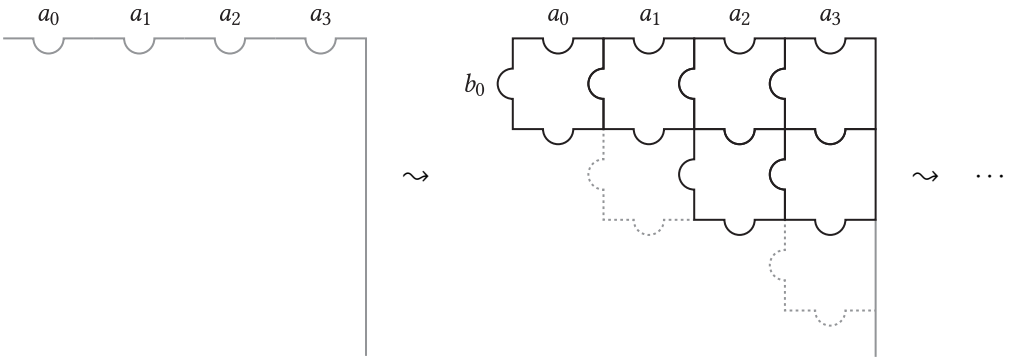
In Nakano's [2013] model, a computation transforms a List A to a CoList B, and to program its behaviour, we need to provide a suitable function $piece : A \times B \to B \times A$. Nakano neatly visualises his model as a jigsaw puzzle. The *piece* function can be thought of as describing a set of jigsaw

pieces (which are not to be rotated or flipped):



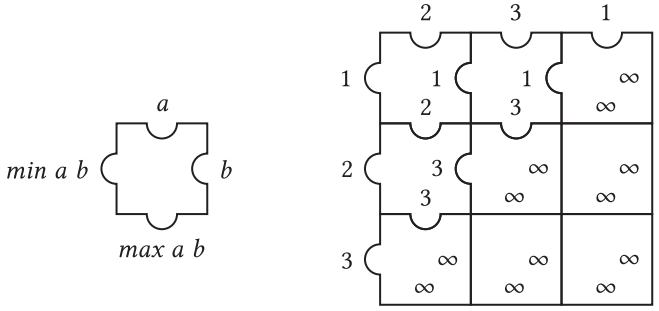
In each piece, the horizontal edges are associated with a value of type A, and the vertical edges with a value of type B. Two pieces fit together exactly when the values on their adjacent edges coincide. Moreover, the values on the top and right edges should determine those on the left and bottom edges, and the *piece* function records the mappings for all the pieces — the piece above, for example, corresponds to the mapping $piece\ (a\,,\ b) = (b'\,,\ a')$.

Below is an illustration of an ongoing computation:



Given an input list, say $a_0 :: a_1 :: a_2 :: a_3 :: []$, we start from an empty board with its top boundary initialised to the input elements and its right boundary to some special "straight" value. Then we put in pieces to fill the board: Whenever a top edge and an adjacent right edge is known, we consult the *piece* function to find the unique fitting piece and put it in. Initially the only place we can put in a piece is the top-right corner, but as we put in more pieces, the number of choices will increase — in the board on the right, for example, we can choose one of the two dashed places to put in the next piece. Eventually we will reach the left boundary, and the values on the left boundary are the output elements. Although we can put in the pieces in a nondeterministic order and even in parallel, the final board configuration is determined by the initial boundary, and thus the output elements are produced deterministically.

▶ *Heapsort in the jigsaw model.* For a concrete example, the heapsort metamorphism can be computed in the jigsaw model with $piece\ (a\,,\ b) = (min\ a\ b\,,\ max\ a\ b)$. The final board configuration after sorting the list $2 :: 3 :: 1 :: []$ is shown below:



(3)

Nakano [2013] remarks that this transforms heapsort into "a form of parallel bubble sort", which looks very different from the original metamorphic computation — in particular, heaps are nowhere to be seen.

In general, how is the jigsaw model related to metamorphisms, and under what conditions does the jigsaw model compute metamorphisms? Again, we will figure out the answers by trying to program jigsaw computations with metamorphic types in Agda.

6.2 The Infinite Case

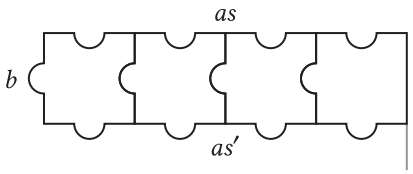
Let us first look at a simpler case where the output colist is always infinite; that is, the coalgebra used in the metamorphic type is just $\circ\, g^\infty$ where $g^\infty : S \to B \times S$ — for heapsort, g^∞ is an adapted version of *popMin* such that popping from the empty heap returns ∞ and the empty heap itself, so the output colist is the sorted input list followed by an infinite number of ∞'s.

6.2.1 *Horizontal Placement.*

We start by giving the metamorphic type:

```
jigsaw_IH : {s : S} → AlgList A (◁) e s → CoalgList B (just ∘ g^∞) s
jigsaw_IH as AlgList A (◁) e s  =  {CoalgList B (just ∘ g^∞) s}₀
```

One possible placement strategy is to place one row of jigsaw pieces at a time. Placing a row is equivalent to transforming an input list *as* into a new one *as′* and also a vertical edge *b*:



We therefore introduce the following function $fill_{\mathrm{IH}}$ for filling a row:[11]

```
fill_IH : {s : S} → AlgList A (◁) e s → B × Σ[ t ∈ S ] AlgList A (◁) e t
fill_IH as  =  {B × Σ[ t ∈ S ] AlgList A (◁) e t}₁
```

We do not know (or cannot easily specify) the index *t* in the type of the new AlgList, so the index is simply existentially quantified. The job of $jigsaw_{\mathrm{IH}}$, then, is to call $fill_{\mathrm{IH}}$ repeatedly to cover the board. We revise Goal 0 into:

```
jigsaw_IH : {s : S} → AlgList A (◁) e s → CoalgList B (just ∘ g^∞) s
decon (jigsaw_IH as AlgList A (◁) e s) with fill_IH as
decon (jigsaw_IH as) | b , t , as′  =  b ::⟨ {just (g^∞ s) ≡ just (b , t)}₂ ⟩ jigsaw_IH as′
```

Goal 2 demands an equality linking *s* and *t*, which are the input and output indices of $fill_{\mathrm{IH}}$. This suggests that $fill_{\mathrm{IH}}$ is responsible for not only computing *t* but also establishing the relationship between *t* and *s*. We therefore add the equality to the result type of $fill_{\mathrm{IH}}$, and discharge Goal 2 with the equality proof that will be produced by $fill_{\mathrm{IH}}$:

```
fill_IH : {s : S} → AlgList A (◁) e s → Σ[ b ∈ B ] Σ[ t ∈ S ] AlgList A (◁) e t × (g^∞ s ≡ (b , t))
fill_IH as  =  {Σ[ b ∈ B ] Σ[ t ∈ S ] AlgList A (◁) e t × (g^∞ s ≡ (b , t))}₁
jigsaw_IH : {s : S} → AlgList A (◁) e s → CoalgList B (just ∘ g^∞) s
decon (jigsaw_IH as) with fill_IH as
decon (jigsaw_IH as) | b , _ , as′ , eq  =  b ::⟨ cong just eq ⟩ jigsaw_IH as′
```

(The combinator '*cong* just' has type $\{X : \mathrm{Set}\}\ \{x\ x' : X\} \to x \equiv x' \to \mathrm{just}\ x \equiv \mathrm{just}\ x'$.)

[11] $\Sigma[\, x \in X \,]\ P\ x$ is the dependent pair type from the Agda Standard Library.

▶ *The road not taken.* From Goal 2, there seems to be another way forward: the equality says that the output vertical edge b and the index t in the type of as' are determined by $g^\infty\ s$, so $jigsaw_{\mathrm{IH}}$ could have computed $g^\infty\ s$ and obtained b and t directly! However, recall that the characteristic of the jigsaw model is that computation proceeds by converting input list elements directly into output colist elements without involving the states appearing in the metamorphic specification. In our setting, this means that states only appear in the function types, not the function bodies, so having $jigsaw_{\mathrm{IH}}$ invoke $g\ s$ would deviate from the jigsaw model. Instead, $jigsaw_{\mathrm{IH}}$ invokes $fill_{\mathrm{IH}}$, which will only use *piece* to compute b. (It would probably be better if we declared the argument s in the metamorphic type as irrelevant to enforce the fact that s does not participate in the computation; this irrelevance declaration would then need to be propagated to related parts in AlgList and CoalgList, though, which we are trying to avoid.)

▶ *Filling a row.* Let us get back to $fill_{\mathrm{IH}}$ (Goal 1). The process of filling a row follows the structure of the input list, so overall it is an induction, of which the first step is a case analysis:

```
fillIH : {s : S} → AlgList A (◁) e s → Σ[ b ∈ B ] Σ[ t ∈ S ] AlgList A (◁) e t × (g∞ s ≡ (b , t))
fillIH []       = {Σ[ b ∈ B ] Σ[ t ∈ S ] AlgList A (◁) e t × (g∞ e ≡ (b , t))}₃
fillIH (a :: as) = {Σ[ b ∈ B ] Σ[ t ∈ S ] AlgList A (◁) e t × (g∞ (a ◁ s) ≡ (b , t))}₄
                   (AlgList (◁) e s)
```

If the input list is empty (Goal 3), we return the rightmost "straight" edge. We therefore assume that a **constant** $straight : B$ is available, and fill it into Goal 3:

```
fillIH : {s : S} → AlgList A (◁) e s → Σ[ b ∈ B ] Σ[ t ∈ S ] AlgList A (◁) e t × (g∞ s ≡ (b , t))
fillIH []       = straight , {Σ[ t ∈ S ] AlgList A (◁) e t × (g∞ e ≡ (straight , t))}₅
fillIH (a :: as) = {Σ[ b ∈ B ] Σ[ t ∈ S ] AlgList A (◁) e t × (g∞ (a ◁ s) ≡ (b , t))}₄
                   (AlgList (◁) e s)
```

At Goal 5, we should now give the new list (along with the index in its type), which we know should have the same length as the old list, so in this case it is easy to see that the new list should be empty as well (and we leave the index in the type of the new list for Agda to infer by giving an underscore):

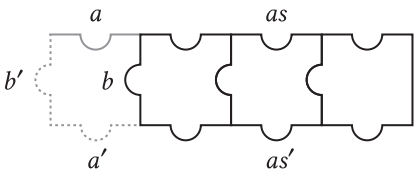
```
fillIH : {s : S} → AlgList A (◁) e s → Σ[ b ∈ B ] Σ[ t ∈ S ] AlgList A (◁) e t × (g∞ s ≡ (b , t))
fillIH []       = straight , _ , [] , {g∞ e ≡ (straight , e)}₆
fillIH (a :: as) = {Σ[ b ∈ B ] Σ[ t ∈ S ] AlgList A (◁) e t × (g∞ (a ◁ s) ≡ (b , t))}₄
                   (AlgList (◁) e s)
```

Here we arrive at another proof obligation (Goal 6), which says that from the initial state e the coalgebra g^∞ should produce *straight* and leave the state unchanged. This seems a reasonable property to add as a condition of the algorithm: in heapsort, for example, e is the empty heap and *straight* is ∞, and popping from the empty heap, as we mentioned, can be defined to return ∞ and the empty heap itself. We therefore add an additional **constant** $straight\text{-}production : g^\infty\ e \equiv (straight\ ,\ e)$, which discharges Goal 6.

The interesting case is when the input list is non-empty (Goal 4). We start with an inductive call to $fill_{\mathrm{IH}}$ itself:

```
fillIH : {s : S} → AlgList A (◁) e s → Σ[ b ∈ B ] Σ[ t ∈ S ] AlgList A (◁) e t × (g∞ s ≡ (b , t))
fillIH []       = straight , _ , [] , straight-production
fillIH (a :: as) with fillIH as
fillIH (a :: as)      | b , s' , as' , eq =
  {Σ[ b ∈ B ] Σ[ t ∈ S ] AlgList A (◁) e t × (g∞ (a ◁ s) ≡ (b , t))}₇
                   (AlgList (◁) e s)
```

With the inductive call, the jigsaw pieces below the tail as have been placed, yielding a vertical edge b and a list as' of horizontal edges below as:



We should complete the row by placing the last jigsaw piece with a and b as input, and use the output edges $(b',\ a') = piece\ (a,\ b)$ in the right places:

$$fill_{\mathrm{IH}} : \{s : S\} \to \mathsf{AlgList}\ A\ (\lhd)\ e\ s \to \Sigma[\, b \in B\,]\ \Sigma[\, t \in S\,]\ \mathsf{AlgList}\ A\ (\lhd)\ e\ t \times (g^{\infty}\ s \equiv (b,\ t))$$
$$fill_{\mathrm{IH}}\ [] = straight\ \_\ []\ straight\text{-}production$$
$$fill_{\mathrm{IH}}\ (a :: as\ \underbrace{\phantom{\mathsf{AlgList}\ (\lhd)\ e\ s}}_{\mathsf{AlgList}\ (\lhd)\ e\ s})\ \mathbf{with}\ fill_{\mathrm{IH}}\ as$$
$$fill_{\mathrm{IH}}\ (a :: as\ \underbrace{\phantom{\mathsf{AlgList}\ (\lhd)\ e\ s}}_{})\ |\ (b,\ s',\ as',\ eq\ \underbrace{\phantom{g^{\infty}\ s \equiv (b,\ s')}}_{g^{\infty}\ s \equiv (b,\ s')}) =$$
$$\mathbf{let}\ (b',\ a') = piece\ (a,\ b)\ \mathbf{in}\ b',\ \_,\ a' :: as',\ \{g^{\infty}\ (a \lhd s) \equiv (b',\ a' \lhd s')\}_8$$

▶ *The jigsaw condition.* Here we see a familiar pattern: Goal 8 demands an equality about producing from a state after consumption, and in the context we have an equality *eq* about producing from a state before consumption. Following what we did in Section 5, a commutative state transition diagram can be drawn:

$$\begin{array}{ccc} & \xleftarrow{\text{consume } a \text{ using } (\lhd)} & \\ a \lhd s & & s \\ \Big\downarrow{\scriptstyle \text{produce } b' \text{ using } g^{\infty}} & \Leftarrow & \Big\downarrow{\scriptstyle \text{produce } b \text{ using } g^{\infty}} \\ a' \lhd s' & & s' \\ & \xleftarrow{\text{consume } a' \text{ using } (\lhd)} & \end{array} \tag{4}$$

where $(b',\ a') = piece\ (a,\ b)$. This is again a kind of commutativity of production and consumption, but unlike the streaming condition (2) in Section 5, the elements produced and consumed can change after swapping the order of production and consumption. Given any top and right edges a and b, the *piece* function should be able to find the left and bottom edges b' and a' to complete the commutative diagram. This constitutes a specification for *piece*, and, inspired by Nakano [2013] (but not following him strictly), we call it the *jigsaw condition*:

constant
$$jigsaw\text{-}condition_{\mathrm{I}} : \{a : A\}\ \{b : B\}\ \{s\ s' : S\} \to$$
$$g^{\infty}\ s \equiv (b,\ s') \to \mathbf{let}\ (b',\ a') = piece\ (a,\ b)\ \mathbf{in}\ g^{\infty}\ (a \lhd s) \equiv (b',\ a' \lhd s')$$

Adding $jigsaw\text{-}condition_{\mathrm{I}}$ as the final assumption, we can fill $jigsaw\text{-}condition_{\mathrm{I}}\ eq$ into Goal 8 and complete the program, which is shown in Figure 3.

▶ *Metamorphisms and the jigsaw model.* We can now see a connection between metamorphic computations and the jigsaw model. Definitionally, a metamorphism folds the input list to a state, and then produces the output elements while updating the state. In the jigsaw model, and with the horizontal placement strategy, rather than folding the input list to a "compressed" state, we use the whole list as an "uncompressed" state, and ensure that the production process using uncompressed states simulates the definitional one using compressed states. The type of $fill_{\mathrm{IH}}$ makes this clear: The produced element b is exactly the one that would have been produced from the compressed state s obtained by folding the old list. Then, on the compressed side, the state s is updated to t;

```
module Jigsaw-Infinite-Horizontal
  ((◁) : A → S → S) (e : S) (g∞ : S → B × S)
  (piece : A × B → B × A)
  (straight : B) (straight-production : g∞ e ≡ (straight , e))
  (jigsaw-condition_I : {a : A} {b : B} {s s′ : S} →
                        g∞ s ≡ (b , s′) → let (b′ , a′) = piece (a , b) in g∞ (a ◁ s) ≡ (b′ , a′ ◁ s′))
  where
  fill_IH : {s : S} → AlgList A (◁) e s → Σ[ b ∈ B ] Σ[ t ∈ S ] AlgList A (◁) e t × (g∞ s ≡ (b , t))
  fill_IH [] = straight , _ , [] , straight-production
  fill_IH (a :: as) with fill_IH as
  fill_IH (a :: as) | b , _ , as′ , eq = let (b′ , a′) = piece (a , b)
                                         in b′ , _ , a′ :: as′ , jigsaw-condition_I eq
  jigsaw_IH : {s : S} → AlgList A (◁) e s → CoalgList B (just ∘ g∞) s
  decon (jigsaw_IV as) with fill_IH as
  decon (jigsaw_IV as) | b , _ , as′ , eq = b ::⟨ cong just eq ⟩ jigsaw_IV as′
```

Fig. 3. Metamorphisms in the infinite jigsaw model with the horizontal placement strategy

correspondingly, on the uncompressed side, the old list is updated to a new list that folds to t. The jigsaw condition ensures that this relationship between compressed and uncompressed states can be maintained by placing rows of jigsaw pieces.

▶ *Deriving the piece function for heapsort using the jigsaw condition.* For the heapsort metamorphism, consuming an element is pushing it into a heap, and producing an element is popping a minimum element from a heap. In diagram (4), producing b on the right means that b is a minimum element in the heap s, and s' is the rest of the heap. If a is pushed into s, popping from the updated heap $a \triangleleft s$ will either still produce b if $a > b$, or produce a if $a \leqslant b$, so b' should be $min\ a\ b$. Afterwards, the final heap $a' \triangleleft s'$ should still contain the other element that was not popped out, i.e., $max\ a\ b$, and can be obtained by pushing $max\ a\ b$ into s', so a' should be $max\ a\ b$.

6.2.2 Vertical Placement. There is another obvious placement strategy: the vertical one, where we place one column of jigsaw pieces at a time. Programming the horizontal placement strategy led us to an understanding of the relationship between metamorphisms and the jigsaw model, and it is likely that programming the vertical placement strategy will lead us to a different perspective. Starting from exactly the same type as $jigsaw_{IH}$:

```
jigsaw_IV : {s : S} → AlgList A (◁) e s → CoalgList B (just ∘ g∞) s
jigsaw_IV as {AlgList A (◁) e s} = {CoalgList B (just ∘ g∞) s}₀
```

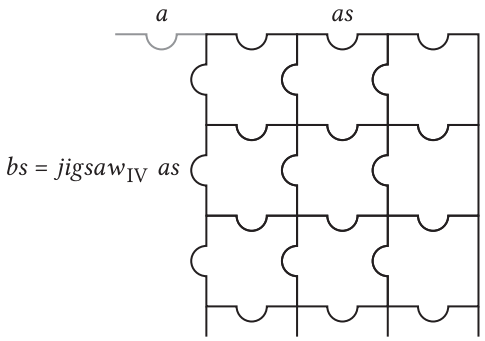
With the vertical placement strategy, we place columns of jigsaw pieces following the structure of the input list as, so we proceed with a case analysis on as:

```
jigsaw_IV : {s : S} → AlgList A (◁) e s → CoalgList B (just ∘ g∞) s
jigsaw_IV [] = {CoalgList B (just ∘ g∞) e}₁
jigsaw_IV (a :: as {AlgList A (◁) e s}) = {CoalgList B (just ∘ g∞) (a ◁ s)}₂
```

If the input list is empty (Goal 1), we should produce a colist of *straight* egdes:

```
jigsaw_IV : {s : S} → AlgList A (◁) e s → CoalgList B (just ∘ g^∞) s
decon (jigsaw_IV []) = straight ::⟨ {just (g^∞ e) ≡ just (straight , e)}_3 ⟩ jigsaw_IV []
jigsaw_IV (a :: as AlgList A (◁) e s) = {CoalgList B (just ∘ g^∞) (a ◁ s)}_2
```

and the proof obligation (Goal 3) is discharged with *cong* just *straight-production*, where both *straight* and *straight-production* are constants we introduced when programming the horizontal placement strategy (Section 6.2.1). For the inductive case (Goal 2):



We place all the columns below the tail *as* by an inductive call $jigsaw_{\mathrm{IV}}\ as$, which gives us a colist of vertical edges. To the left of this colist, we should place the last column below the head element *a*; again we introduce a helper function $fill_{\mathrm{IV}}$ that takes *a* and the colist $jigsaw_{\mathrm{IV}}\ as$ as input and produces the leftmost colist:

```
jigsaw_IV : {s : S} → AlgList A (◁) e s → CoalgList B (just ∘ g^∞) s
decon (jigsaw_IV []) = straight ::⟨ cong just straight-production ⟩ jigsaw_IV []
jigsaw_IV (a :: as) = fill_IV a (jigsaw_IV as)
```

▶ *Filling a column.* Upon receiving the helper type command (`C-c C-h`), Agda again can give us a suitable type of $fill_{\mathrm{IV}}$:

```
fill_IV : {s : S} (a : A) → CoalgList B (just ∘ g^∞) s → CoalgList B (just ∘ g^∞) (a ◁ s)
fill_IV a bs CoalgList B (just ∘ g^∞) s = {CoalgList B (just ∘ g^∞) (a ◁ s)}_4
```

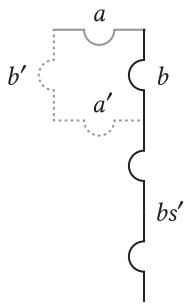
Here we should deconstruct *bs* so that we can invoke *piece* on *a* and the first element of *bs*:

```
fill_IV : {s : S} (a : A) → CoalgList B (just ∘ g^∞) s → CoalgList B (just ∘ g^∞) (a ◁ s)
decon (fill_IV a bs CoalgList B (just ∘ g^∞) s) with decon bs
decon (fill_IV a bs) | ⟨ eq just (g^∞ s) ≡ nothing ⟩ = {CoalgListF B (just ∘ g^∞) (a ◁ s)}_5
decon (fill_IV a bs) | b ::⟨ eq just (g^∞ s) ≡ just (b , s') ⟩ bs' CoalgList B (just ∘ g^∞) s' =
                       {CoalgListF B (just ∘ g^∞) (a ◁ s)}_6
```

For Goal 5, since the coalgebra just ∘ g^∞ in the type of *bs* never returns nothing, it is impossible for *bs* to be empty. We can convince Agda that this case is impossible by matching *eq* with the absurd pattern (), saying that *eq* cannot possibly exist (and Agda accepts this because a just-value can never be equal to nothing):

```
fill_IV : {s : S} (a : A) → CoalgList B (just ∘ g^∞) s → CoalgList B (just ∘ g^∞) (a ◁ s)
decon (fill_IV a bs CoalgList B (just ∘ g^∞) s) with decon bs
decon (fill_IV a bs) | ⟨ () ⟩
decon (fill_IV a bs) | b ::⟨ eq just (g^∞ s) ≡ just (b , s') ⟩ bs' CoalgList B (just ∘ g^∞) s' =
                       {CoalgListF B (just ∘ g^∞) (a ◁ s)}_6
```

The real work is done at Goal 6, where bs is deconstructed into its head b and tail bs':



We invoke the *piece* function to transform a and b into b' and a'; the head of the output colist is then b', and the tail is coinductively computed from a' and bs':

```
fill_IV : {s : S} (a : A) → CoalgList B (just ∘ g^∞) s → CoalgList B (just ∘ g^∞) (a ◁ s)
decon (fill_IV a bs [CoalgList B (just ∘ g^∞) s]) with decon bs
decon (fill_IV a bs) | ⟨ () ⟩
decon (fill_IV a bs) | b ::⟨ eq [just (g^∞ s) ≡ just (b , s')] ⟩ bs' [CoalgList B (just ∘ g^∞) s'] =
  let (b' , a') = piece (a , b) in b' ::⟨ {just (g^∞ (a ◁ s)) ≡ just (b' , a' ◁ s')}_7 ⟩ fill_IV a' bs'
```

The remaining proof obligation is exactly the jigsaw condition (4) modulo the harmless occurrences of just, so we arrive at the complete program shown in Figure 4, which uses the same set of conditions as the horizontal placement strategy.

▶ *Metamorphisms and the jigsaw model revisited.* Now we can see that the vertical placement strategy indeed corresponds to another way of thinking about metamorphic computations in the jigsaw model. In contrast to streaming metamorphisms (Section 5), where we need to be cautious about producing an element because once an element is produced we can no longer change it, computing metamorphisms in the jigsaw model with the vertical placement strategy is like having an entire output colist right from the start and then updating it:

- initially we start with a colist of *straight* edges, which is unfolded from the empty state e;
- inductively, if we have a colist unfolded from some state s, and an input element a comes in, we place a column of jigsaw pieces to update the colist, and the result — due to the jigsaw condition — is a colist unfolded from the new state $a \triangleleft s$;
- finally, after all elements of the input list as are consumed, we get a colist unfolded from $foldr\ (\triangleleft)\ e\ as$.

Notably, the inductive step is faithfully described by the type of $fill_{\mathrm{IV}}$ (which was generated by Agda).

6.3 The General (Possibly Finite) Case

Finally, let us tackle the general case, where the produced colist can be finite. This is the same setting as Nakano's [2013], and will allow us to compare our derived conditions with his. The metamorphic type we use is exactly the one we saw in Section 3:

```
jigsaw : {s : S} → AlgList A (◁) e s → CoalgList B g s
jigsaw as [AlgList A (◁) e s] = {CoalgList B g s}_0
```

We use the vertical placement strategy, so the overall structure will be similar to $jigsaw_{\mathrm{IV}}$ (Section 6.2.2/Figure 4). Starting from a case analysis:

```
module Jigsaw-Infinite-Vertical
  ((◁) : A → S → S) (e : S) (g^∞ : S → B × S)
  (piece : A × B → B × A)
  (jigsaw-condition_I : {a : A} {b : B} {s s′ : S} →
                        g^∞ s ≡ (b , s′) → let (b′ , a′) = piece (a , b) in g^∞ (a ◁ s) ≡ (b′ , a′ ◁ s′))
  (straight : B) (straight-production : g^∞ e ≡ (straight , e))
  where

  fill_IV : {s : S} (a : A) → CoalgList B (just ∘ g^∞) s → CoalgList B (just ∘ g^∞) (a ◁ s)
  decon (fill_IV a bs) with decon bs
  decon (fill_IV a bs) | ⟨ () ⟩
  decon (fill_IV a bs) | b ::⟨ eq ⟩ bs′ =
    let (b′ , a′) = piece (a , b)
    in b′ ::⟨ cong just (jigsaw-condition_I (cong-from-just eq)) ⟩ fill_IV a′ bs′
         -- cong-from-just : {X : Set} {x x′ : X} → just x ≡ just x′ → x ≡ x′

  jigsaw_IV : {s : S} → AlgList A (◁) e s → CoalgList B (just ∘ g^∞) s
  decon (jigsaw_IV []) = straight ::⟨ cong just straight-production ⟩ jigsaw_IV []
  jigsaw_IV (a :: as) = fill_IV a (jigsaw_IV as)
```

Fig. 4. Metamorphisms in the infinite jigsaw model with the vertical placement strategy

```
  jigsaw : {s : S} → AlgList A (◁) e s → CoalgList B g s
  jigsaw [] = {CoalgList B g e}_1
  jigsaw (a :: as [AlgList A (◁) e s]) = {CoalgList B g (a ◁ s)}_2
```

At Goal 1, it should suffice to produce an empty colist:

```
  jigsaw : {s : S} → AlgList A (◁) e s → CoalgList B g s
  decon (jigsaw []) = ⟨ {g e ≡ nothing}_3 ⟩
  jigsaw (a :: as [AlgList A (◁) e s]) = {CoalgList B g (a ◁ s)}_2
```

To do so we need $g\ e \equiv \text{nothing}$, which is a reasonable assumption — for heapsort, for example, e is the empty heap, on which *popMin* computes to nothing. We therefore introduce a **constant** *nothing-from-e* : $g\ e \equiv \text{nothing}$ and use it to discharge Goal 3:

```
  jigsaw : {s : S} → AlgList A (◁) e s → CoalgList B g s
  decon (jigsaw []) = ⟨ nothing-from-e ⟩
  jigsaw (a :: as [AlgList A (◁) e s]) = {CoalgList B g (a ◁ s)}_2
```

For Goal 2, we proceed in exactly the same way as we dealt with the corresponding case of $jigsaw_{\text{IV}}$:

```
  jigsaw : {s : S} → AlgList A (◁) e s → CoalgList B g s
  decon (jigsaw []) = ⟨ nothing-from-e ⟩
  jigsaw (a :: as) = fill a (jigsaw as)
```

where the type and the top-level structure of the helper function *fill* is also exactly the same as $fill_{\text{IV}}$:

```
  fill : {s : S} (a : A) → CoalgList B g s → CoalgList B g (a ◁ s)
  decon (fill a bs [CoalgList B g s]) with decon bs
```

```
decon (fill a bs) | ⟨ eq _ : g s ≡ nothing ⟩ = {CoalgListF B g (a ◁ s)}₄
decon (fill a bs) | b ::⟨ eq _ : g s ≡ just (b , s') ⟩ bs' : CoalgList B g s' = {CoalgListF B g (a ◁ s)}₅
```

The situation gets more interesting from this point.

▶ *Filling a column.* Let us work on the familiar case first, namely Goal 5. If we do the same thing as the corresponding case of $fill_{\mathrm{IV}}$:

```
fill : {s : S} (a : A) → CoalgList B g s → CoalgList B g (a ◁ s)
decon (fill a bs : CoalgList B g s) with decon bs
decon (fill a bs) | ⟨ eq _ : g s ≡ nothing ⟩ = {CoalgListF B g (a ◁ s)}₄
decon (fill a bs) | b ::⟨ eq _ : g s ≡ just (b , s') ⟩ bs' : CoalgList B g s' =
  let (b' , a') = piece (a , b) in b' ::⟨ {g (a ◁ s) ≡ just (b' , a' ◁ s')}₆ ⟩ fill a' bs'
```

we will see that the condition we need is depicted in the same way as the diagram (4) for the infinite jigsaw condition. Formally it is slightly different though, because we need to wrap the results of g in the just constructor:

```
{a : A} {b : B} {s s' : S} →
g s ≡ just (b , s') → let (b' , a') = piece (a , b) in g (a ◁ s) ≡ just (b' , a' ◁ s'))   (5)
```

We will come back to this condition and close Goal 6 later.

Goal 4, unlike the corresponding case of $fill_{\mathrm{IV}}$, is no longer an impossible case. We might be tempted to produce an empty colist here:

```
fill : {s : S} (a : A) → CoalgList B g s → CoalgList B g (a ◁ s)
decon (fill a bs : CoalgList B g s) with decon bs
decon (fill a bs) | ⟨ eq _ : g s ≡ nothing ⟩ = ⟨ {g (a ◁ s) ≡ nothing}₇ ⟩
decon (fill a bs) | b ::⟨ eq _ : g s ≡ just (b , s') ⟩ bs' : CoalgList B g s' =
  let (b' , a') = piece (a , b) in b' ::⟨ {g (a ◁ s) ≡ just (b' , a' ◁ s')}₆ ⟩ fill a' bs'
```

But the proof obligation indicates that this is not a right choice. Let us call a state s "empty" exactly when $g\ s \equiv \mathsf{nothing}$. The proof obligation says that if a state s is empty then the next state $a \triangleleft s$ should also be empty, but obviously this does not hold in general. For heapsort, pushing a finite element to a heap always results in a non-empty heap, constituting a counterexample. On the other hand, it is conceivable that we can make some elements satisfy this property — for example, it is reasonable to define the *push* operation on heaps such that pushing ∞ into an empty heap keeps the heap empty — so producing an empty colist is not always wrong.

▶ *Flat elements.* The above reasoning suggests that we should do a case analysis on a to determine whether to produce an empty or non-empty colist at Goal 4. Let us call an element "flat" exactly when subsuming it into an empty state results in another empty state. We should be given a decision procedure $flat^?$ that can be used to identify flat elements:

```
constant flat? : (a : A) → ({s : S} → g s ≡ nothing → g (a ◁ s) ≡ nothing) ⊎ {Set}₈
```

Traditionally, $flat^?$ would return a boolean, but using booleans in dependently typed programming is almost always a "code smell" since their meaning — for example, whether the input satisfies a certain property or not — will almost always need to be explained to the type-checker later; instead, it is more convenient to make the decision procedure directly return a proof or a refutation of the property. In the case of $flat^?$, its type directly says that an element of A is flat or otherwise. This "otherwise" at Goal 8 also requires some thought. We could fill in the negation of the "flat" property, but it may turn out that we need something stronger. Unable to decide now, let us leave Goal 8 open for the moment, and come back when we have more information.

Abandoning Goal 7, we roll back to Goal 4 and refine it into Goals 9 and 10:

```
fill : {s : S} (a : A) → CoalgList B g s → CoalgList B g (a ◁ s)
decon (fill a bs CoalgList B g s ) with decon bs
decon (fill a bs) | ⟨ eq g s ≡ nothing ⟩ with flat? a
decon (fill a bs) | ⟨ eq ⟩ | inj₁ flat       = ⟨ {g (a ◁ s) ≡ nothing}₉ ⟩
decon (fill a bs) | ⟨ eq ⟩ | inj₂ not-flat  = {CoalgListF B g (a ◁ s)}₁₀
decon (fill a bs) | b ::⟨ eq g s ≡ just (b , s') ⟩ bs' CoalgList B g s'  =
  let (b' , a') = piece (a , b) in b' ::⟨ {g (a ◁ s) ≡ just (b' , a' ◁ s')}₆ ⟩ fill a' bs'
```

At Goal 9, we know that a is flat, so it is fine to produce an empty colist; the proof obligation is easily discharged with *flat eq*, where *flat* is the proof given by $flat^?$ affirming that a is flat.

For Goal 10, we want to invoke *piece* and produce a non-empty colist. However, the input colist is empty, so we do not have a vertical input edge for *piece*. The situation is not entirely clear here, but let us make some choices first and see if they make sense later. Without an input vertical edge, let us again introduce a **constant** $straight : B$, which solves the problem about using *piece*. Also, in the coinductive call that generates the tail, we use *bs* (the only colist available in the context) as the second argument:

```
fill : {s : S} (a : A) → CoalgList B g s → CoalgList B g (a ◁ s)
decon (fill a bs CoalgList B g s ) with decon bs
decon (fill a bs) | ⟨ eq g s ≡ nothing ⟩ with flat? a
decon (fill a bs) | ⟨ eq ⟩ | inj₁ flat       = ⟨ flat eq ⟩
decon (fill a bs) | ⟨ eq ⟩ | inj₂ not-flat  =
  let (b' , a') = piece (a , straight) in b' ::⟨ {g (a ◁ s) ≡ just (b' , a' ◁ s)}₁₁ ⟩ fill a' bs
decon (fill a bs) | b ::⟨ eq g s ≡ just (b , s') ⟩ bs' CoalgList B g s'  =
  let (b' , a') = piece (a , b) in b' ::⟨ {g (a ◁ s) ≡ just (b' , a' ◁ s')}₆ ⟩ fill a' bs'
```

▶ *The jigsaw condition.* Now let us examine whether our choices are sensible. The expected type at Goal 11 can be depicted as:

$$\begin{array}{ccc} a \lhd s & \xleftarrow{\text{consume } a \text{ using } f} & s \\ \Big\downarrow{\scriptstyle \text{produce } b' \text{ using } g} & & \Big\downarrow{\scriptstyle \text{(produce } stright \text{ using } g)} \\ a' \lhd s & \xleftarrow[\text{consume } a' \text{ using } f]{} & s \end{array} \tag{6}$$

The dashed transition on the right is not a real state transition — we know that s is an empty state since in the context we have $eq : g\ s \equiv \text{nothing}$. Completing the above diagram (6) with the dashed transition allows us to compare it with the diagram (4) for the infinite jigsaw condition, and the key to the comparison is to link the notions of empty states in the infinite case and the general (possibly finite) case. In the infinite case, we have a condition $straight\text{-}production : g^\infty\ e \equiv (straight\ ,\ e)$ saying that the *straight* edge is produced from the empty state e, which remains unchanged after production. We could have defined empty states in the infinite case to be the states s such that $g^\infty\ s \equiv (straight\ ,\ s)$ (although this was not necessary). Now, the general (possibly finite) case can be thought of as an optimisation of the infinite case. We stop producing *straight* edges from empty states — that is, we modify the coalgebra to return nothing from empty states — because these *straight* edges provide no information: if we omit, and only omit, the production of these *straight* edges, then whenever a vertical input edge is missing we know that it can only be *straight*.

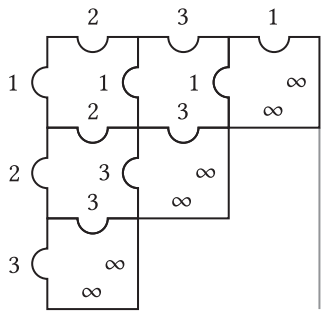
However, the modification to the coalgebra destroys the production transitions from empty states in the infinite jigsaw condition. What remains is condition (5), and cases involving empty states and *straight* edges as depicted by diagram (6) above are left out.

One thing we can do is to merge diagram (6) back into condition (5) by relaxing the latter's premise:

```
constant
  jigsaw-condition :
    {a : A} {b : B} {s s′ : S} →
    g s ≡ just (b , s′) ⊎ (g s ≡ nothing × g (a ◁ s) ≢ nothing × b ≡ straight × s′ ≡ s) →
    let (b′ , a′) = piece (a , b) in g (a ◁ s) ≡ just (b′ , a′ ◁ s′)
```

Note that we include $g\ (a \triangleleft s) \not\equiv \text{nothing}$ in the new part of the premise to rule out the case where a is flat. A proof of this type should come from $flat^?$, so at Goal 8 we make $flat^?$ return a proof of $\{s : S\} \to g\ (a \triangleleft s) \not\equiv \text{nothing}$ when the input element a is not flat. Finally, having *jigsaw-condition* in the context is informative enough for Auto to discharge both Goals 6 and 11 for us, and we arrive at the complete program in Figure 5.

▶ *Comparison with Nakano's [2013] jigsaw condition.* How do our conditions compare with Nakano's [2013, Definition 5.1]? Ours seem to be weaker, but this is probably because our algorithm is not as sophisticated as it could be. Nakano imposes three conditions, which he refers to collectively as the jigsaw condition: the first one is exactly our *nothing-from-e*, the second one is related to flat elements but more complicated than our corresponding formulation, and the third one, though requiring some decoding, is almost our *jigsaw-condition*. Comparing Nakano's third condition with our *jigsaw-condition* reveals that there was one possibility that we did not consider: at Goal 5 we went ahead and produced a non-empty colist, but producing an empty colist was also a possibility. Our current *jigsaw* algorithm places columns of non-decreasing lengths from right to left like:



If we performed some case analysis at Goal 5 like for Goal 4, we might have been able to come up with an algorithm that could decrease column lengths when going left, saving more jigsaw pieces (although probably not for heapsort).

7 DISCUSSION

We end our experiment at this point, but apparently there are more stories to be told. It was slightly mysterious that we chose to implement the streaming algorithm with a left metamorphic type and the jigsaw model with a right metamorphic type — is this pairing necessary or coincidental? It turns out that the jigsaw model also works for left metamorphisms, and we even get a slightly more general algorithm if we start from a left metamorphic type — this we left as an exercise for the reader. (One solution is included in the supplementary code.) Streaming, on the other hand, seems to be inherently associated with left metamorphisms. One way to explain this might be

```
module Jigsaw-General
  ((◁) : A → S → S) (e : S) (g : S → Maybe (B × S))
  (piece : A × B → B × A)
  (straight : B)
  (flat? : (a : A) → ({s : S} → g s ≡ nothing → g (a ◁ s) ≡ nothing) ⊎
                     ({s : S} → g (a ◁ s) ≢ nothing))
  (nothing-from-e : g e ≡ nothing)
  (jigsaw-condition :
    {a : A} {b : B} {s s′ : S} →
    g s ≡ just (b , s′) ⊎ (g s ≡ nothing × g (a ◁ s) ≢ nothing × b ≡ straight × s′ ≡ s) →
    let (b′ , a′) = piece (a , b) in g (a ◁ s) ≡ just (b′ , a′ ◁ s′))
  where
  fill : {s : S} (a : A) → CoalgList B g s → CoalgList B g (a ◁ s)
  decon (fill a bs) with decon bs
  decon (fill a bs) | ⟨ eq ⟩ with flat? a
  decon (fill a bs) | ⟨ eq ⟩ | inj₁ flat     = ⟨ flat eq ⟩
  decon (fill a bs) | ⟨ eq ⟩ | inj₂ not-flat =
    let (b′ , a′) = piece (a , straight)
    in b′ ::⟨ jigsaw-condition (inj₂ (eq , not-flat , refl , refl)) ⟩ fill a′ bs
  decon (fill a bs) | b ::⟨ eq ⟩ bs′ =
    let (b′ , a′) = piece (a , b)
    in b′ ::⟨ jigsaw-condition (inj₁ eq) ⟩ fill a′ bs′
  jigsaw : {s : S} → AlgList A (◁) e s → CoalgList B g s
  decon (jigsaw []) = ⟨ nothing-from-e ⟩
  jigsaw (a :: as)  = fill a (jigsaw as)
```

Fig. 5. Metamorphisms in the general (possibly finite) jigsaw model (with the vertical placement strategy)

as follows: During streaming, we consume an initial segment of an input list, pause to do some production, and then resume consumption of the rest of the list. The natural way to do this is to consume the list from the left, examining and removing head elements and keeping the tail for the resumption of production. That is, we are really treating the input list as a finite colist. This suggests that a streaming algorithm in general should be a transformation from colists to colists, both possibly infinite — that is, it is in general a "stream processor" [Ghani et al. 2009]. We might also consider using the jigsaw model to implement stream processors, but the situation can be more complicated: if we think of computation in the jigsaw model as updating the output colist (Section 6.2.2), in general the output colist can change forever so that we are never sure whether any of the output elements has stabilised. More fundamentally, the current metamorphic specification — a fold followed by an unfold — is no longer adequate: since the input colist can be infinite, we have to replace the fold with some infinite consumption process, meaning that in general we can no longer reach an intermediate state and switch to production. We have to go back to the basics and think about how we might specify the behaviour of stream processors, and tell a new story from there.

We should say that this paper is mostly a faithful report of the actual developments of the programs from their specifications (as types). There are only minor deviations from the actual developments for streamlining the presentation, and apart from the general ideas of streaming and the jigsaw model, few hints were taken from the original papers by Gibbons [2007] and Nakano [2013]. For example, from Goal 0 to Goals 1 and 2 in Section 5, bringing in decon and *inspect* was really a decision that could only be made later (at Goal 3). And a step that might have been influenced by Gibbons was the leap from diagram (1) to diagram (2), i.e., Gibbons's streaming condition, but Agda did take us to diagram (1), which was probably close enough to the streaming condition. We certainly did not rely on any of the proofs in the original papers, simply because we did not have to construct most of them. For example, if we compare our development of the streaming algorithm (Section 5) with that of Bird and Gibbons [2003] (who gave the original formulation and proof of the streaming theorem), we will see that their Lemma 29 turns into our *streaming-lemma* and their Theorem 30 goes implicitly into the typing of *stream* and no longer needs special attention. Notably, our formal treatment of the jigsaw model is independent and evidently different from Nakano's (which is somewhat obscure) — the whole Section 6 can be seen as an attempt to understand the relationship between metamorphisms and the jigsaw model without looking into Nakano's proofs.

So what does the experiment tell us about interactive type-driven development? Fundamentally, our development is just the familiar mathematical activity of formulating conditions of theorems by trying to prove the theorems and finding out what is missing. Agda does make the process unusually smooth, though. In addition to the automation provided by Agda, we speculate that the smoothness is due to what we might call "proof locality", by which we mean that proofs appear near where they matter. As a result, the programmer gains better "situation awareness" of what a program means while constructing the program. This was helpful when, for example, we saw at Goal 7 in Section 6.3 that we did not make a right choice because the obligation was too strong for the premise we had, and avoided going down the wrong path. By contrast, with the extrinsic approach, proofs are formally constructed only later and separately, and it would be harder for the programmer to develop the same level of "situation awareness", especially without the help of a proof assistant.

The experiment is far from decisive when it comes to determining the effectiveness of the paradigm of interactive type-driven development, though. As mentioned in the beginning, we deliberately chose a problem that was not too complicated to make the experiment more likely to succeed. If we aim for a more sophisticated problem, for example, "metaphorisms" [Oliveira 2018], which are metamorphisms whose result is optimised in some way, it will probably require more advanced techniques like that of Ko and Gibbons's [2013], where a type is first transformed into one that is closer to program structure. We should be doing a lot more experiments with interactive type-driven development, and success with more sophisticated algorithmic problems just might be a boost that the paradigm needs.

ACKNOWLEDGEMENTS

The author thanks Shin-Cheng Mu for a discussion during ICFP 2017 in Oxford, and Jeremy Gibbons for discussions in Oxford and Tokyo, and for reading and commenting on a draft of this paper. This work originated from the author's DPhil work at Oxford [Ko 2014], which was supported by a University of Oxford Clarendon Scholarship and the UK Engineering and Physical Sciences Research Council (EPSRC) project *Reusability and Dependent Types*. After the author moved to NII, the work continued with the support of the Japan Society for the Promotion of Science (JSPS) Grant-in-Aid for Scientific Research (A) No. 25240009 and (S) No. 17H06099.

REFERENCES

Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming Infinite Structures by Observations. In *Symposium on Principles of Programming Languages (POPL'13)*. ACM. https://doi.org/10.1145/2429069.2429075

Richard Bird and Jeremy Gibbons. 2003. Arithmetic Coding with Folds and Unfolds. In *Advanced Functional Programming (Lecture Notes in Computer Science)*, Vol. 2638. Springer, 1–26. https://doi.org/10.1007/978-3-540-44833-4_1

Edwin Brady. 2013. Idris, a General-Purpose Dependently Typed Programming Language: Design and Implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

Neil Ghani, Peter Hancock, and Dirk Pattinson. 2009. Representations of Stream Processors Using Nested Fixed Points. *Logical Methods in Computer Science* 5, 3:9 (2009). https://doi.org/10.2168/LMCS-5(3:9)2009

Jeremy Gibbons. 2007. Metamorphisms: Streaming Representation-Changers. *Science of Computer Programming* 65, 2 (2007), 108–139. https://doi.org/10.1016/j.scico.2006.01.006

Hsiang-Shang Ko. 2014. *Analysis and Synthesis of Inductive Families*. DPhil thesis. University of Oxford. https://ora.ox.ac.uk/objects/ora:9019

Hsiang-Shang Ko and Jeremy Gibbons. 2013. Relational Algebraic Ornaments. In *Workshop on Dependently Typed Programming (DTP'13)*. ACM, 37–48. https://doi.org/10.1145/2502409.2502413

Conor McBride. 2011. Ornamental Algebras, Algebraic Ornaments. (2011). https://personal.cis.strath.ac.uk/conor.mcbride/pub/OAAO/LitOrn.pdf

Keisuke Nakano. 2013. Metamorphism in Jigsaw. *Journal of Functional Programming* 23, 2 (2013), 161–173. https://doi.org/10.1017/S0956796812000391

Ulf Norell. 2007. *Towards a Practical Programming Language based on Dependent Type Theory*. Ph.D. Dissertation. Chalmers University of Technology. http://www.cse.chalmers.se/~ulfn/papers/thesis.html

José Nuno Oliveira. 2018. Programming from Metaphorisms. *Journal of Logical and Algebraic Methods in Programming* 94 (2018), 15–44. https://doi.org/10.1016/j.jlamp.2017.09.003