JFP **31**, e15, 15 pages, 2021.
doi:10.1017/S0956796821000113

EDUCATION MATTERS

How to design co-programs

JEREMY GIBBONS

Department of Computer Science, University of Oxford, Oxford, UK
(*e-mail:* jeremy.gibbons@cs.ox.ac.uk)

Abstract

The observation that *program structure follows data structure* is a key lesson in introductory programming: good hints for possible program designs can be found by considering the structure of the data concerned. In particular, this lesson is a core message of the influential textbook "How to Design Programs" by Felleisen, Findler, Flatt, and Krishnamurthi. However, that book discusses using only the structure of *input* data for guiding program design, typically leading towards structurally *recursive* programs. We argue that novice programmers should also be taught to consider the structure of *output* data, leading them also towards structurally *corecursive* programs.

1 Introduction

Where do programs come from?

This mystery can be an obstacle to novice programmers, who can become overwhelmed by the design choices presented by a blank sheet of paper, or an empty editor window—where does one start? A good place to start, we tell them, is by analyzing the structure of the data that the program is to consume. For example, if the program h is to process a list of values, one may start by analyzing the structure of that list. Either the list is empty ($[\,]$), or it is non-empty ($a : x$) with a head (a) and a tail (x). This provides a candidate program structure:

$$\begin{array}{lcl} h\,[\,] & = & \ldots \\ h\,(a : x) & = & \ldots\, a \ldots x \ldots \end{array}$$

where for the empty list some result must simply be chosen, and for a non-empty list the result depends on the head a and tail x. Moreover, x is another list, because lists are a recursively structured datatype; and it is often the case that the list-consuming program h can be structured with the same shape as the datatype, making a recursive call on x:

$$\begin{array}{lcl} h\,[\,] & = & \ldots \\ h\,(a : x) & = & \ldots\, a \ldots x \ldots h\, x \ldots \end{array}$$

A key lesson in teaching introductory programming is therefore that *program structure follows data structure*. This lesson is a central message in the influential textbook "How to Design Programs" (Felleisen *et al.*, 2001, 2004, 2018) (henceforth "HtDP"). This book is organized around *design recipes*, one of which is for structural recursion as above.

However, there is more structure available to guide the novice programmer in designing their program than simply the structure of the input data; in particular, there is also the structure of the *output* data. For a program h that produces a list, we have an alternative candidate program structure determined by the output, which is either empty ($[\,]$), or non-empty ($b : y$) with a head (b) and a tail (y):

$$\begin{array}{lll} h\,x & | & \ldots x \ldots \;=\; [\,] \\ & | & \mathit{otherwise} \;=\; b : y \\ & \textbf{where} & b \;=\; \ldots x \ldots \\ & & y \;=\; \ldots x \ldots \end{array}$$

Again, the tail y is another list, and is often generated by a recursive call of h:

$$\begin{array}{lll} h\,x & | & \ldots x \ldots \;=\; [\,] \\ & | & \mathit{otherwise} \;=\; b : y \\ & \textbf{where} & b \;=\; \ldots x \ldots \\ & & y \;=\; h\,(\ldots x \ldots) \end{array}$$

This program structure is *corecursive*, designed around the structure of the output rather than the structure of the input. The input x need provide no guidance; indeed, it might be absent altogether—for example, in a corecursive definition of the list consisting of the first 1,000 prime numbers, which is not a function at all.

HtDP presents no design recipe corresponding to this dual structure, and neither (to the best of this author's knowledge) does any other introductory programming textbook. The thesis of this paper is that this dual design recipe is just as important, and novice programmers deserve also to be taught *how to design co-programs*.

2 Design recipes

One key aspect of HtDP is the emphasis on *design recipes* for solving programming tasks. A design recipe is a process for solving a programming problem, constructing a sequence of specific, checkable products along the way: a contract for the function, analogous to a type signature (but HtDP treats types informally, so this signature is just a comment); a statement of purpose; a function header; example inputs and outputs; and a template for the function body. Following the design recipe entails filling in a particular contract etc, then fleshing out the function body from its template, and finally testing the resulting program against the initial examples.

The primary strategy for problem solving in the book is via analysis of the structure of the input. When the input is composite, like a record, the template should name the available fields as likely ingredients of the solution. When the input has "mixed data", such as a union type, the template should enumerate the alternatives, leading to a case analysis in the solution. When the input is of a recursive type, the template encapsulates *structural recursion*—a case analysis between base cases and inductive cases, the latter entailing recursive calls. The design recipe for structural recursion (Felleisen *et al.*, 2001, Figure 26) is shown in Figure 1.

One of HtDP's illustrations of structural recursion is sorting (Felleisen *et al.*, 2001, Chapter 12). Simply following the design recipe for the purpose of sorting (and following

https://doi.org/10.1017/S0956796821000113 Published online by Cambridge University Press

Phase	Goal	Activity
data analysis and design	to formulate a data definition	develop a data definition for mixed data with at least two alternatives; one alternative must not refer to the definition; explicitly identify all self-references in the data definition

contract, purpose, and header	to name the function; to specify its classes of input data and its class of output data; to describe its purpose; to formulate a header	name the function, the classes of input data, the class of output data, and specify its purpose:
		`;; name : in1 in2 ... --> out`
		`;; to compute ... from x1 ...`
		`(define (name x1 x2 ...) ...)`

examples	to characterize the input–output relationship via examples	create examples of the input–output relationship; make sure there is at least one example per subclass
template	to formulate an outline	develop a cond-expression with one clause per alternative; add selector expressions to each clause; annotate the body with natural recursions; test: the self-references in this template and the data definition match!
body	to define the function	formulate a Scheme expression for each simple cond-line; explain for all other cond-clauses what each natural recursion computes according to the purpose statement
test	to discover mistakes ("typos" and logic)	apply the function to the inputs of the examples; check that the outputs are as predicted

Fig. 1. The design recipe for structural recursion.

it a second time for insertion into an ordered list) leads inexorably to the discovery of Insertion Sort.

Let us retrace those steps. The *data definition* for the inputs (and as it happens, also for the outputs) is that of lists of integers: a list is either empty ([ ]), or non-empty ($a : x$) with a head (a, an integer) and a tail (x, another list). The *contract* and *header* in HtDP amount to a type declaration for the function:

$$insertSort :: [Integer] \to [Integer]$$

The *purpose* is to sort the input list of numbers; HtDP chooses to sort in descending order, but we will sort in ascending order. The design recipe calls for some *examples*, with at least one example in which the input is the empty list, and at least one in which it is a non-empty list:

$$\begin{aligned}
insertSort\,[\,] &= [\,] \\
insertSort\,[3,2,1] &= [1,2,3] \\
insertSort\,[1,2,3] &= [1,2,3] \\
insertSort\,[12,20,-5] &= [-5,12,20] \\
insertSort\,[1,1,2] &= [1,1,2]
\end{aligned}$$

The *template* instantiates the generic program structure for this particular function:

$$\begin{aligned}
insertSort\,[\,] &= \ldots \\
insertSort\,(a : x) &= \ldots a \ldots x \ldots insertSort\,x \ldots
\end{aligned}$$

https://doi.org/10.1017/S0956796821000113 Published online by Cambridge University Press

We then have to fill in the *body* to complete the definition. The base case is easy:

$$insertSort\,[\,] \;=\; [\,]$$

For the inductive step, HtDP introduces a *wish list* including an auxilliary function *insert*:

$$insertSort\,(a : x) \;=\; insert\,a\,(insertSort\,x)$$

so the function recurses on the tail of a non-empty list $a : x$, inserting the head a into the sorted subresult $insertSort\,x$. HtDP then walks through the same sequence of steps to implement the wished-for auxilliary function:

$$\begin{array}{l} insert :: Integer \to [Integer] \to [Integer] \\ insert\,b\,[\,] \;=\; [b] \\ insert\,b\,(a : x) \;\mid\; b \leqslant a \;=\; b : a : x \\ \qquad\qquad\;\; \mid\; b > a \;=\; a : insert\,b\,x \end{array}$$

The overall program satisfies the invariant that the second argument to *insert* is sorted, and therefore so is the result of *insert*. It should finally be *tested*. From now on, we elide most of the design recipe steps; apart from the template, they are no different here from those in HtDP.

A secondary, more advanced, strategy in HtDP is to use *generative recursion*, in which the recursive calls are on subproblems that are generated from the input by some more significant computation than simply projecting out a component. Instances include `while` loops and divide and conquer. The template in this design recipe incorporates a test for triviality; in the nontrivial cases, it splits the problem into one or more subproblems, recursively solves the subproblems, and assembles the subresults into an overall result.

One of the motivating examples for generative recursion is Quick Sort (Felleisen *et al.*, 2001, Section 25.2), albeit not Hoare's fast in-place version—dividing a non-empty input list into two parts using the head as the pivot, recursively sorting both parts, and concatenating the results with the pivot in the middle:

$$\begin{array}{l} quickSort :: [Integer] \to [Integer] \\ quickSort\,x \;\mid\; null\,x \;=\; [\,] \\ \qquad\qquad\;\; \mid\; otherwise \;=\; quickSort\,y +\!\!+ [head\,x] +\!\!+ quickSort\,z \\ \quad \textbf{where}\;\; y \;=\; [b \mid b \leftarrow tail\,x, b \leqslant head\,x] \\ \qquad\qquad z \;=\; [b \mid b \leftarrow tail\,x, b > head\,x] \end{array}$$

No other program structures than structural recursion and generative recursion are considered in HtDP. (Other design recipes are considered, in particular accumulating parameters and imperative features. But these do not determine the gross structure of the resulting program. Moreover, the Second Edition (Felleisen *et al.*, 2018) drops the imperative recipe.)

3 Co-programs

The thesis of this paper is that HtDP has missed an opportunity to reinforce its core message, that *program structure follows data structure*. Specifically, some programs have a shape determined by the shape of the input, but others have a shape determined instead by

https://doi.org/10.1017/S0956796821000113 Published online by Cambridge University Press

the shape of the output. In particular, the next design recipes to consider after structural recursion, before getting to generative recursion, should be for *constructing compound output* and *structural corecursion*.

More concretely, a function that generates "mixed output"—whether that is a union type, or simply a boolean—might be defined by case analysis over the output. A function that generates a record might be composed of subprograms that generate each of the fields of that record. A function that generates a recursive data structure from some input data might be defined with a case analysis as to whether the result is trivial, and for nontrivial cases with recursive calls to generate substructures of the result. Novice programmers should be given explicit design recipes to address these possibilities, as they are for program structures determined by the input data.

For an example of mixed output, consider a program that may fail, such as division. One technique for handling failure is to guard the definition in order to return an alternative value in cases that would otherwise fail:

$$safeDiv :: Integer \to Integer \to Maybe\ Integer$$

A value of type *Maybe Integer* is either *Just n* for some integer *n*, or *Nothing*; a function returning a *Maybe* will make a case analysis leading to one or other of those two outcomes. There should be a design recipe applicable to programs returning a *Maybe*.

Whereas, the design recipe for structural recursion calls for examples that cover the all possible *input* variants, examples for co-programs should cover all possible *output* variants. So a minimal collection of examples for *safeDiv* is:

$$\begin{aligned} safeDiv\ 7\ 2 &= Just\ 3 \\ safeDiv\ 7\ 0 &= Nothing \end{aligned}$$

The design recipe should provide a template, which when instantiated to *safeDiv* is:

$$\begin{aligned} safeDiv\ x\ y \mid & \ldots x \ldots y \ldots = Nothing \\ \mid & \ otherwise \ = Just\ (\ldots x \ldots\ y \ldots) \end{aligned}$$

Then considering the given examples should lead directly to the expected final program:

$$\begin{aligned} safeDiv\ x\ y \mid & \ y == 0 \ = Nothing \\ \mid & \ otherwise \ = Just\ (x\ `div`\ y) \end{aligned}$$

Now, certainly this program performs a case analysis, and of course the analysis depends on the input data. But the analysis is not determined by the *structure* of the input, only its *value*—in particular, whether *y* is zero. So the best explanation of the program structure is not that it is determined by the structure of the input, but that it is determined by the structure of the output.

For an example of generating composite output, consider the problem of extracting a date, represented as a record:

$$\textbf{data}\ Date = Date\ \{day :: Day, month :: Month, year :: Year\}$$

from a formatted string. The template is naturally structured to match the output type:

$$\begin{aligned} &readDate :: String \to Date \\ &readDate\ s \ = \ Date\ \{day = d, month = m, year = y\} \end{aligned}$$

https://doi.org/10.1017/S0956796821000113 Published online by Cambridge University Press

$$\begin{array}{lrl} \textbf{where} & d & = \ldots s \ldots \\ & m & = \ldots s \ldots \\ & y & = \ldots s \ldots \end{array}$$

The output consists of several components, *day*, *month*, *year*; so one should consider a program structure with subprograms d, m, y, each generating one of those components.

For an example of corecursion, consider the problem of "zipping" together two input lists to a list of pairs—taking $[1,2,3]$ and $[4,5,6,7]$ to $[(1,4),(2,5),(3,6)]$, and pruning the result to the length of the shorter input. One can again solve the problem by case analysis on the input, but the fact that there are two inputs makes that a bit awkward—whether to do case analysis on one list in favour of the other, or to analyze both at once. For example, here is the outcome of a case analysis on the first input, followed—if the first input is non-empty—by a case analysis on the second:

$$\begin{array}{lll} zip :: [\alpha] \to [\beta] \to [(\alpha, \beta)] & & \\ zip\ [\,]\ \ y & = & [\,] \\ zip\ (a : x)\ \ [\,] & = & [\,] \\ zip\ (a : x)\ \ (b : y) & = & (a, b) : zip\ x\ y \end{array}$$

The more direct case analysis on both inputs at once would lead to four cases rather than three, which would not be an improvement.

One can instead solve the problem by case analysis on the output—and it is arguably more natural to do so, because there is only one output rather than two. For a structurally corecursive program towards lists, there are three questions to ask:

1. *When is the output empty?*
2. *If the output isn't empty, what is its head?*
3. *And from what data is its tail recursively constructed?*

These questions are analogous to the "question-and-answer games" introduced in the Second Edition (Felleisen *et al.*, 2018, Figures 52, 53). The answers for *zip* are:

1. *When is the output empty?* — When either input is empty.
2. *If the output isn't empty, what is its head?* — The pair of input heads.
3. *And from what data is its tail recursively constructed?* — The pair of input tails.

which leads the following program:

$$\begin{array}{lll} zip :: [\alpha] \to [\beta] \to [(\alpha, \beta)] & & \\ zip\ x\ y \ \mid \ null\ x \vee null\ y & = & [\,] \\ \quad \mid \ otherwise & = & (head\ x, head\ y) : zip\ (tail\ x)\ (tail\ y) \end{array}$$

(This version of *zip* is written using list destructors *null*, *head*, *tail*. It is possible to write it instead with list constructors $[\,]$, $(:)$ as patterns, and arguably more idiomatic Haskell to do so; but we will use destructors for corecursion to emphasize the distinction from recursion.)

Sorting provides another example of corecursion. Whereas Insertion Sort is a structural recursion over the input list, inserting elements one by one into a sorted intermediate result, *Selection Sort* is a structural corecursion towards the output list, repeatedly extracting the minimum remaining element as the next element of the output. As a question-and-answer game:

https://doi.org/10.1017/S0956796821000113 Published online by Cambridge University Press

1. *When is the output empty?* — When the input is empty.
2. *If the output isn't empty, what is its head?* — The minimum of the input.
3. *And from what data is the tail recursively generated?* — The input without this minimum element.

Simply following the structural corecursion design recipe for the purpose of sorting leads inexorably to the discovery of Selection Sort:

$$
\begin{array}{l}
\mathit{selectSort} :: [\mathit{Integer}] \to [\mathit{Integer}] \\
\mathit{selectSort}\ x \mid \mathit{null}\ x \quad = [\,] \\
\qquad\qquad \mid \mathit{otherwise} = \mathbf{let}\ a = \mathit{minimum}\ x\ \mathbf{in}\ a : \mathit{selectSort}\ (x \setminus\setminus [a])
\end{array}
$$

(here, $x \setminus\setminus y$ denotes list x with the elements of list y removed).

4 Generative recursion

Only once this dual form of program structure has been explored should students be encouraged to move on to generative recursion, because the latter exploits both structural recursion and structural corecursion. For example, the Quick Sort algorithm that is used as the main motivating example of generative recursion (Felleisen *et al.*, 2001, Section 25.2)

$$
\begin{array}{l}
\mathit{quickSort} :: [\mathit{Integer}] \to [\mathit{Integer}] \\
\mathit{quickSort} = \mathit{flatten} \cdot \mathit{build}
\end{array}
$$

essentially consists of a structural corecursion *build* to construct an intermediate tree,

$$
\begin{array}{l}
\mathbf{data}\ \mathit{NTree} = \mathit{Empty} \mid \mathit{Node}\ \mathit{NTree}\ \mathit{Integer}\ \mathit{NTree} \\
\mathit{build} :: [\mathit{Integer}] \to \mathit{NTree} \\
\mathit{build}\ x \mid \mathit{null}\ x \quad = \mathit{Empty} \\
\qquad\quad \mid \mathit{otherwise} = \mathit{Node}\ (\mathit{build}\ y)\ (\mathit{head}\ x)\ (\mathit{build}\ z) \\
\quad \mathbf{where}\ y = [b \mid b \leftarrow \mathit{tail}\ x, b \leqslant \mathit{head}\ x] \\
\qquad\quad\ z = [b \mid b \leftarrow \mathit{tail}\ x, b > \mathit{head}\ x]
\end{array}
$$

followed by structural recursion *flatten* over that tree to produce the resulting list:

$$
\begin{array}{l}
\mathit{flatten} :: \mathit{NTree} \to [\mathit{Integer}] \\
\mathit{flatten}\ \mathit{Empty} \quad = [\,] \\
\mathit{flatten}\ (\mathit{Node}\ t\ a\ u) = \mathit{flatten}\ t +\!\!+ [a] +\!\!+ \mathit{flatten}\ u
\end{array}
$$

Both *build* and *flatten* have a binary pattern of recursive calls; they evidently are structured according to neither the input nor the output, which are both lists. Instead, they follow the structure of the intermediate *NTree* datatype, with *build* as structural corecursion and *flatten* as structural recursion. Some insight is still required to come up with the intermediate datatype, because it is not explicit in the problem statement; but at least now there is a concrete artifact for the insight to aim for.

A similar explanation applies to any divide-and-conquer algorithm. For example, consider Merge Sort, which is another divide-and-conquer sorting algorithm:

$$
\begin{array}{l}
\mathit{mergeSort} :: [\mathit{Integer}] \to [\mathit{Integer}] \\
\mathit{mergeSort} = \mathit{mergeAll} \cdot \mathit{splitUp}
\end{array}
$$

https://doi.org/10.1017/S0956796821000113 Published online by Cambridge University Press

It can be implemented following the same intermediate tree shape, but this time with a simple splitting phase:

$$
\begin{array}{l}
splitUp :: [Integer] \to NTree \\
splitUp\ x \mid null\ x \quad = Empty \\
\qquad\quad \mid otherwise \;=\; \mathbf{let}\ (y,z) = halve\ (tail\ x)\ \mathbf{in} \\
\qquad\qquad\qquad\quad\ \ Node\ (splitUp\ y)\ (head\ x)\ (splitUp\ z)
\end{array}
$$

$$
\begin{array}{l}
halve :: [\alpha] \to ([\alpha],[\alpha]) \\
halve\ [\,] \quad = ([\,],[\,]) \\
halve\ [a] \quad = ([a],[\,]) \\
halve\ (a:b:x) \;=\; \mathbf{let}\ (y,z) = halve\ x\ \mathbf{in}\ (a:y, b:z)
\end{array}
$$

and all the comparisons in the recombining phase:

$$
\begin{array}{l}
mergeAll :: NTree \to [Integer] \\
mergeAll\ Empty \quad = [\,] \\
mergeAll\ (Node\ t\ a\ u) \;=\; merge\ (mergeAll\ t)\ (merge\ [a]\ (mergeAll\ u))
\end{array}
$$

$$
\begin{array}{l}
merge :: [Integer] \to [Integer] \to [Integer] \\
merge\ [\,]\ \ y \quad = y \\
merge\ x \quad [\,] \quad = x \\
merge\ (a:x)\ \ (b:y) \;=\; \mathbf{if}\ a \leqslant b\ \mathbf{then}\ a : merge\ x\ (b:y)\ \mathbf{else}\ b : merge\ (a:x)\ y
\end{array}
$$

As it turns out, using the same tree type for Merge Sort as for Quick Sort is a bit clunky, on account of the two calls to *merge* required in *mergeAll*. It is neater to use a different kind of tree, namely non-empty externally labelled binary trees, with elements at the leaves and none at the branches:

$$\mathbf{data}\ BTree \;=\; Tip\ Integer \mid Bin\ BTree\ BTree$$

Because *BTree* accommodates only non-empty trees, we should use *Maybe BTree* as the intermediate datatype, with *Nothing* for the empty list and *Just* for a non-empty list. This leads to the following program:

$$
\begin{array}{l}
mergeSort_2 :: [Integer] \to [Integer] \\
mergeSort_2\ x \;=\; mergeAll_2\ (splitUp_2\ x)
\end{array}
$$

$$
\begin{array}{l}
splitUp_2 :: [Integer] \to Maybe\ BTree \\
splitUp_2\ x \mid null\ x \quad = Nothing \\
\qquad\quad \mid single\ x \quad = Just\ (Tip\ (head\ x)) \\
\qquad\quad \mid otherwise \;=\; \mathbf{let}\ (y,z) = halve\ x \quad \text{-- } x \text{ has length at least 2} \\
\qquad\qquad\qquad\qquad\quad\ \ (Just\ t, Just\ u) = (splitUp_2\ y, splitUp_2\ z) \\
\qquad\qquad\qquad\quad\ \ \mathbf{in}\ Just\ (Bin\ t\ u)
\end{array}
$$

$$
\begin{array}{l}
single :: [\alpha] \to Bool \\
single\ [a] \;=\; True \\
single\ x \quad = False
\end{array}
$$

$$
\begin{array}{l}
mergeAll_2 :: Maybe\ BTree \to [Integer] \\
mergeAll_2\ Nothing \quad = [\,] \\
mergeAll_2\ (Just\ (Tip\ a)) \quad = [a] \\
mergeAll_2\ (Just\ (Bin\ t\ u)) \;=\; merge\ (mergeAll_2\ (Just\ t))\ (mergeAll_2\ (Just\ u))
\end{array}
$$

https://doi.org/10.1017/S0956796821000113 Published online by Cambridge University Press

Now there are three cases to consider: no elements, one element, and two or more elements, corresponding to the intermediate data *Nothing*, *Just* (*Tip a*), *Just* (*Bin t u*), respectively.

The clunkiness of the first *mergeSort* is a learning opportunity: to realize that there is a problem, to come up with a fix (tip labels rather than node labels in the tree, so non-empty trees, so use *Maybe*), rearrange the furniture accordingly, and then replay the development and compare the results.

Having identified structural recursion and structural corecursion as separate parts, they may now be studied separately; separation of concerns is a crucial lesson in introductory programming. Moreover, the parts may be put together in different ways. The divide-and-conquer pattern is known in the "mathematics of program construction" community as a *hylomorphism* (Meijer *et al.*, 1991), an unfold to generate a call tree followed by a fold to consume that tree. As the Quick Sort example suggests, the tree can always be *deforested* (Wadler, 1990)—it is a *virtual data structure* (Swierstra & de Moor, 1993). But the converse pattern, of a fold from some structured input to some intermediate value, followed by an unfold to a different structured output, is also important—it can be seen as a change of structured representation, and has been called a *metamorphism* (Gibbons, 2007). One simple application is to convert a number from an input base (a sequence of digits in that base), via an intermediate representation (the represented number), to an output base (a different sequence of digits). This time, the intermediate data is unstructured, and the two phases follow the structure of the input and output types. More interesting applications include encoding and data compression algorithms, such as *arithmetic coding* (Bird & Gibbons, 2003) and *asymmetric numeral systems* (Gibbons, 2019).

Intriguingly, the illustration of generative recursion immediately preceding Quick Sort in HtDP changes between editions of the book. In the First Edition (Felleisen *et al.*, 2001, Section 25.1), the example is a simple `while` loop, moving a ball at constant speed across a table until it drops over an edge:

$$\begin{array}{lll} \mathit{moveUntilOut}\ \mathit{ball} \mid \mathit{outOfBounds}\ \mathit{ball} & = & \mathit{ball} \\ \quad\quad\quad\quad\quad\quad \mid \mathit{otherwise} & = & \mathit{moveUntilOut}\ (\mathit{moveBall}\ \mathit{ball}) \end{array}$$

The problem of tracking the ball is trivial if the ball is already out of bounds; otherwise, move the ball a little in its current direction, and repeat. But in the Second Edition (Felleisen *et al.*, 2018, Section 25.1), the problem is changed to bundling a sequence up into chunks of a given size:

$$\mathit{bundle}\ 3\ \texttt{"abcdefg"} = [\texttt{"abc"}, \texttt{"def"}, \texttt{"g"}]$$

The book considers structural recursion, but concludes that "a structural approach cannot work". Indeed, structural recursion does not work; but structural *corecursion* works beautifully:

1. *When is the output empty?* — When the input list is empty.
2. *If the output isn't empty, what is its head?* — The first *n*-chunk of the input, or the whole input if it is too short.
3. *And from what data is the tail recursively generated?* — All but the first *n*-chunk.

https://doi.org/10.1017/S0956796821000113 Published online by Cambridge University Press

It turns out that this is a classical example of corecursion:

$$\begin{array}{l} bundle :: Int \to [\alpha] \to [[\alpha]] \\ bundle\ n\ x \ \mid null\ x \quad\quad = [\,] \\ \qquad\qquad \mid otherwise \ = take\ n\ x : bundle\ n\ (drop\ n\ x) \end{array}$$

The corecursive program we end up with is exactly the same as in HtDP. But there is nothing tricky or *ad hoc* about the design process, and no eureka required: the full power of generative recursion is not needed, and the program follows directly from the design recipe for structural corecursion.

5 Laziness

Although we have used Haskell as a notation, nothing above depends on laziness; it would all work as well in ML or Scheme. It is true that the mathematical structures underlying structural recursion and structural corecursion are prettier when one admits infinite data structures—the final coalgebra of the base functor for lists is the datatype of finite *and infinite* lists (Meijer *et al.*, 1991), and without admitting the infinite structures some recursive definitions (such as $ones = 1 : ones$) have no solution. But that sophistication is beyond the scope of introductory programming, in which context it suffices to restrict attention to finite data structures.

HtDP already stipulates a termination argument in the design recipe for generative recursion; the same kind of argument could be required for structural corecursion—and is easy to make for the sorting, zipping, and bundling examples given above. Of course, structural recursion over finite data structures is necessarily terminating. But laziness is unnecessary for co-programming.

6 Structured programming

It is not a new assertion that output structure is important. Ramsey (2014) recorded that his experience in teaching HtDP led him to the following lesson:

> *Last, and rarely, you could design a function's template around the introduction form for the result type. When I teach [HtDP] again, I will make my students aware of this decision point in the construction of a function's template: should they use elimination forms, function composition, or an introduction form?*

Ramsey also elaborates on test coverage:

> *Check functional examples to be sure every choice of input is represented. Check functional examples to be sure every choice of* output *is represented. This activity is especially valuable for functions returning Booleans.*

(his emphasis). Of course, not all functions need be surjective—but predicates presumably should be. We should pay attention to output data structure as well as to input data structure.

https://doi.org/10.1017/S0956796821000113 Published online by Cambridge University Press

In fact, HtDP draws on a long tradition of relating data structure and program structure. Hoare wrote in his "Notes on Data Structuring":

> *There are certain close analogies between the methods used for structuring data and the methods for structuring a program which processes that data. (Hoare, 1972)*

Brooks put it pithily in "The Mythical Man-Month":

> *Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; theyâĂŹll be obvious. (Brooks, 1975)*

which was modernized by Raymond in his essay "The Cathedral and the Bazaar" to:

> *Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious. (Raymond, 1999)*

HtDP credits Jackson Structured Programming as partial inspiration for the design recipe approach. As Jackson wrote:

> *The central theme of this book has been the relationship between data and program structures. The data provides a model of the problem environment, and by basing our program structures on data structures we ensure that our programs will be intelligible and easy to maintain. (Jackson, 1975, p. 279)*

and:

> *The structure of a program must be based on the structures of* all *of the data it processes. (Jackson, 1975, p. 151)*

(my emphasis). In a retrospective lecture, he clarified:

> *program structure should be dictated by the structure of its input and output data streams. (Jackson, 2002)*

Among other classes of problems, Jackson discusses "boundary clashes"—for example, *Telegrams Analysis* (Jackson, 1975, Problem 13), or reformatting paragraphs of text from one page width to another. As a simple instance, consider converting a data stream from 80 columns to a 132 columns, such as from punch card input to line printer output; this is a perfect example of a metamorphism, being a structural corecursion (*bundle* 132) after a structural recursion (*concat*). The JSP approach was designed for processing sequential streams of input records to similar streams of output records, and the essence of the approach is to identify the structures of each of the data files (input and output) in terms of sequences, selections, and iterations, to refine them all to a common structure that matches all of them simultaneously, and to use that common data structure as the program structure. So even way back in 1975 it was clear that we need to pay attention to the structure of output data as well as to that of input data.

https://doi.org/10.1017/S0956796821000113 Published online by Cambridge University Press

7 Discussion

HtDP apologizes that generative recursion

> *is much more of an ad hoc activity than the data-driven design of structurally recursive functions. Indeed, it is almost better to call it inventing an algorithm than designing one. Inventing an algorithm requires a new insight—a "eureka". (Felleisen et al., 2001, Chapter 25)*

It goes on to suggest that mere programmers cannot generally be expected to have such algorithmic insights:

> *In practice, new complex algorithms are often developed by mathematicians and mathematical computer scientists; programmers, though, must thoroughly understand the underlying ideas so that they can invent the simple algorithms on their own and communicate with scientists about the others. (Felleisen et al., 2001, Chapter 25)*

This defeatism is a consequence of not following through on the core message, that program structure follows data structure. Quick Sort and Merge Sort are not *ad hoc*: their program structure clearly does follow some data structure. Admittedly, they do require some insight in order to identify structure that is not present in either the input or the output; but having identified that structure, there is no further mystery, and no *ad hockery* required. This is even clearer for *bundle*: the determining data structure is the output type itself, already present in the problem statement.

Presenting both structural recursion and structural corecursion as design schemes may leave open a choice: some problems, like sorting and zipping, are amenable to both approaches, so which does one choose? It might seem at first that providing a choice makes things more difficult—as they say, a person with one clock knows what time it is, but a person with two clocks is never sure. But really, all the choice does is to reveal a question that needs to be answered by some other means. It is not that either approach is inherently better than the other; rather, the programmer should be open to exploring both possibilities, and choosing one only after considering the consequences. Perhaps one leads to a neater program with less duplication of code or computation, or to abstractions or intermediate results that can be reused elsewhere.

8 Conclusion

Once one's eyes have been opened towards them, co-programs start appearing everywhere. As it happens, I have just finishing marking a programming assignment involving a crossword whose entries were Roman numerals; I found two co-programs from students that I had not considered in my model answers. One co-program was for the function that converts a whole number at most 3, 999 to a Roman numeral. My model answer was expressed with structural recursion over the digits of the input number:

```
roman :: Integer → String
roman n  =  consume (labelled n)
  where  consume [ ]          =  ""
```

https://doi.org/10.1017/S0956796821000113 Published online by Cambridge University Press

$$
\begin{array}{lcl}
\mathit{consume}\,((d,p):x) & = & \mathit{digit}\,d\,p +\!\!+ \mathit{consume}\,x \\
\mathit{labelled}\,n & = & \ldots \quad \text{-- eg } \mathit{labelled}\,789 = [(7,2),(8,1),(9,0)] \\
\mathit{digit}\,d\,p & = & \ldots \quad \text{-- eg } \mathit{digit}\,7\,2 = \texttt{"DCC"}
\end{array}
$$

But some students came up instead with a structural corecursion towards chunks of the output numeral:

$$
\begin{array}{lcl}
\mathit{roman}\,n & = & \mathit{concat}\,(\mathit{produce}\,n\,\mathit{letters}) \\
\quad \textbf{where}\ \mathit{produce}\,n\,x & \mid & \mathit{null}\,x \quad = \quad [\,] \\
& \mid & n \geqslant v \quad = \quad r : \mathit{produce}\,(n-v)\,x \\
& \mid & \mathit{otherwise} \quad = \quad \mathit{produce}\,n\,(\mathit{tail}\,x) \\
\qquad \textbf{where}\,(v,r) & = & \mathit{head}\,x \\
\mathit{letters} & = & [(1{,}000, \texttt{"M"}), (900, \texttt{"CM"}), (500, \texttt{"D"}), \ldots]
\end{array}
$$

(What is the first chunk of the output? And from what number should we compute the remaining chunks?)

The other co-program was the function to generate all permutations of a list. My model answer was expressed with structural recursion over the input list:

$$
\begin{array}{lcl}
\mathit{perms} :: [\alpha] \to [[\alpha]] & & \\
\mathit{perms}\,[\,] & = & [[\,]] \\
\mathit{perms}\,(a:x) & = & [y +\!\!+ [a] +\!\!+ z \mid x' \leftarrow \mathit{perms}\,x, (y,z) \leftarrow \mathit{splits}\,x'] \\
\quad \textbf{where}\ \mathit{splits}\,[\,] & = & [([\,],[\,])] \\
\mathit{splits}\,(a:x) & = & (a:x,[\,]) : [(y, a:z) \mid (y,z) \leftarrow \mathit{splits}\,x]
\end{array}
$$

But some students came up instead with a structural corecursion towards the lists in the output collection:

$$
\begin{array}{lcl}
\mathit{perms} :: [\alpha] \to [[\alpha]] & & \\
\mathit{perms}\,x \mid \mathit{null}\,x & = & [[\,]] \\
\mid \mathit{otherwise} & = & [a:z \mid (a,y) \leftarrow \mathit{pick}\,x, z \leftarrow \mathit{perms}\,y] \\
\quad \textbf{where}\ \mathit{pick}\,[\,] & = & [\,] \\
\mathit{pick}\,(a:x) & = & (a,x) : [(b, a:y) \mid (b,y) \leftarrow \mathit{pick}\,x]
\end{array}
$$

(Of what inputs is [ ] a permutation? And for permutations of the form $a : z$, what values can a take? And what can z be a permutation of?)

Admittedly, neither of these examples precisely fits the basic pattern. Nevertheless, the distinction between "program structure follows input data structure" and "program structure follows output data structure" is helpful in characterizing the branching point in the design process.

Novice programmers need guidance on how to design programs; one important lesson is to be led by the structure of the data. Both structural recursion and structural corecursion are tools, and students need as many tools as we can provide. If we provide them only with a hammer, they will naturally treat every problem as a nail; if we provide them with a spanner too, at least we can encourage them to ask themselves whether a particular metal object is more nail-like or more bolt-like. HtDP tells a good story about being led by the structure of the input data, but omits to teach students look also at the structure of the output data. This is a missed opportunity.

https://doi.org/10.1017/S0956796821000113 Published online by Cambridge University Press

Dedication: This paper arose out of a talk I presented at the *Matthias Felleisen Half-Time Show*, a symposium held in Boston in November 2018 in celebration of Matthias's 60th birthday. Matthias is known for many contributions to the field of Programming Languages; he received the ACM Karl V. Karlstrom Outstanding Educator Award in 2009, and the ACM SIGPLAN Programming Languages Achievement Award in 2012. And of course, he was Editor-in-Chief of the *Journal of Functional Programming* for many years.

The basis of Matthias's Karlstrom Award is a long series of collaborative projects on teaching introductory programming, including TeachScheme! (Bloch *et al.*, 1995–2007) Program by Design (Bloch *et al.*, 2009–2012), and HtDP (Felleisen *et al.*, 2001, 2018). Matthias describes himself as an iconoclast, and so I make no apologies for provocative use of Haskell syntax in my examples.

Acknowledgments

The author would like to thank the participants at the Matthias Felleisen Half-Time Show and the anonymous reviewers for their many helpful comments, and especially Shriram Krishnamurthi for his enthusiastic incitement and shepherding of this paper.

Conflicts of Interest

None.

References

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

Bloch, S., Clements, J., Felleisen, M., Findler, R., Fisler, K., Flatt, M., Proulx, V. & Krishnamurthi, S. (1995–2007) *TeachScheme!* https://teach-scheme.org/

Bloch, S., Clements, J., Felleisen, M., Findler, R., Fisler, K., Flatt, M., Proulx, V. & Krishnamurthi, S. (2009–2012) *Program by Design*. https://programbydesign.org/

Brooks, Jr, F. P. (1975) *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley.

Felleisen, M., Findler, R. B., Flatt, M. & Krishnamurthi, S. (2001) *How to Design Programs*, 1st edn. MIT Press. https://htdp.org/2003-09-26/Book/

Felleisen, M., Findler, R. B., Flatt, M. & Krishnamurthi, S. (2004) The structure and interpretation of the computer science curriculum. *J. Funct. Program.* **14**(4), 365–378.

Felleisen, M., Findler, R. B., Flatt, M. & Krishnamurthi, S. (2018) *How to Design Programs*, 2nd edn. MIT Press. https://htdp.org/2018-01-06/Book/

Gibbons, J. (2007) Metamorphisms: Streaming representation-changers. *Science of Computer Programming* **65**(2), 108–139.

Gibbons, J. (2019) Coding with asymmetric numeral systems. In *Mathematics of Program Construction*, Hutton, G. (ed), Lecture Notes in Computer Science, vol. 11825. Springer-Verlag, pp. 444–465.

Hoare, C. A. R. (1972) Notes on data structuring. In *Structured Programming*, Dahl, O.-J., Dijkstra, E. W. & Hoare, C. A. R. (eds). APIC Studies in Data Processing. Academic Press, pp. 83–174.

Jackson, M. A. (1975) *Principles of Program Design*. Academic Press.

https://doi.org/10.1017/S0956796821000113 Published online by Cambridge University Press

Jackson, M. A. (2002) JSP in perspective. In *Software Pioneers: Contributions to Software Engineering*, Broy, M. & Denert, E. (eds). Springer-Verlag, pp. 480–493. Available at: http://mcs.open.ac.uk/mj665/JSPPers1.pdf

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

Ramsey, N. (2014) On teaching "How to Design Programs": Observations from a newcomer. In *International Conference on Functional Programming*. Association for Computing Machinery, p. 153–166.

Raymond, E. S. (1999) *The Cathedral and the Bazaar*. O'Reilly Media.

Swierstra, D. & de Moor, O. (1993) Virtual data structures. *IFIP TC2/WG2.1 State-of-the-Art Report on Formal Program Development*, Möller, B., Partsch, H. and Schuman, S. (eds), Lecture Notes in Computer Science, vol. 755. Springer-Verlag, pp. 355–371.

Wadler, P. (1990) Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science* **73**, 231–248.