

# Thud and Blunder

Conor Mc Bride and ...?

February 5, 2026

## 1 Introduction

This is a positive little story about learning the implementations of recursive functions on inductive data by perturbation testing, provided the functions concerned are sufficiently unremarkable. Let me give you an example. Consider the type  $\text{Tree } X$  of binary *tree expressions* with free variables in  $X$ , generated by  $\bullet$  and  $l \wedge r$  for  $l, r : \text{Tree } X$ . Any pair of

$$\text{leafy} : \text{Tree } \{\} \quad \text{nodey} : \text{Tree } \{\text{ll}, \text{rr}\}$$

generates a simple structurally recursive function

$$\begin{aligned} \mathbf{transform} &: \text{Tree } \{\} \rightarrow \text{Tree } \{\} \\ \mathbf{transform } \bullet &= \text{leafy} \\ \mathbf{transform } (l \wedge r) &= \text{nodey}[\mathbf{transform } l / \text{ll}, \mathbf{transform } r / \text{rr}] \end{aligned}$$

in which *leafy* and *nodey* act as templates, with free variables showing where to substitute the results of the recursive subcomputations. Now, if you have a secret function

$$\mathbf{mystery} : \text{Tree } \{\} \rightarrow \text{Tree } \{\}$$

I have a way to choose candidates for *leafy* and *nodey* which will make **transform** approximate **mystery**, and moreover, if your **mystery** is itself a **transform**, my approximation will behave the same, even if I do not choose exactly the same *leafy* and *nodey* as you. I can get all the information I need by feeding the following four trees (for which I have pet names) to **mystery**:

$\bullet$	‘thud’
$\bullet \wedge \bullet$	‘blunder’
$(\bullet \wedge \bullet) \wedge \bullet$	‘blunderthud’
$\bullet \wedge (\bullet \wedge \bullet)$	‘thudblunder’

Note that ‘blunderthud’ and ‘thudblunder’ are both generated from ‘thud’  $\wedge$  ‘thud’ (also known as ‘blunder’) by replacing exactly one ‘thud’ with a ‘blunder’.

Here is how I do it. First, I choose  $\text{leafy} = \mathbf{mystery } \bullet = \mathbf{mystery } \text{‘thud’}$ . So far, so good. Next, I construct *nodey* by exploring **mystery** (‘thud’  $\wedge$  ‘thud’) to see wherein it has my *leafy* but **mystery** ‘blunderthud’ or from **mystery** ‘thudblunder’ do not. These are the telltale signs that replacing ‘thud’ by ‘blunder’ has made a difference. I place ll to mark discrepancy with **mystery** ‘blunderthud’ and, otherwise, rr to mark discrepancy with **mystery** ‘thudblunder’. I am now sure that my **transform** and your **mystery** agree on both ‘thud’ and ‘blunder’, at least, even if they differ elsewhere: there are plenty of **mystery** functions which are not **transforms**, and they may very well do so.

But I can prove that if your **mystery** is a **transform** and it fails to distinguish ‘thud’ from ‘blunder’, then it is constant, and if it *does* tell them apart, then I shall indeed compute exactly the *leafy* and *nodey* you chose!

## 2 Finite Types, Finitary Containers, Free Monads

Binary trees come in two *shapes*,  $\{\bullet, \wedge\}$  and for each shape we can give the set of their *positions* for substructures — leaves have none and nodes have two:

$$\bullet \mapsto \{\} \quad \wedge \mapsto \{\text{ll}, \text{rr}\}$$

It is reassuring that these types are finite and enumerable. Let us cling to the small. In particular, let us imagine that we have a (small) type  $\mathbf{UF}$  of (descriptions of) finite types. Perhaps we might reify  $\mathbf{UF}$  by means of a universe construction. How we do so is not locally important, but it is interesting to identify requirements for the thing. If we have some  $X : \mathbf{UF}$ , I shall cheerfully take  $X$  to be a type, with no notation for the translation. (What joy to be Russell and Tarski at once!)

I certainly expect to be able to encode enumerations such as  $\{\}$  and  $\{\text{ll}, \text{rr}\}$  in  $\mathbf{UF}$ , but I expect other structure, besides. In particular, I shall need dependent *pair* types<sup>1</sup>:

$$(x : S) \times T[x]$$

which is in  $\mathbf{UF}$  if both  $S$  and  $T[x]$  are.

However, where our types include dependent function types

$$(x : S) \rightarrow T[x]$$

$\mathbf{UF}$  allows us to *tabulate* functions.

$$(x : S) \sqsupset T[x]$$

where  $S$  is in  $\mathbf{UF}$ , meaning that a function can be encoded as a tuple of  $T[x]$ s, and the whole type is in  $\mathbf{UF}$  if  $T[x]$  is. Crucially, if  $S$  is in  $\mathbf{UF}$ , we can enumerate every possible input to such a function and thus visit every possible output.

From these finite types, we can build finitary *containers*  $C : \mathbf{Ctn}$  to express what it is to be a ‘layer of structure’. Such a  $C$  is given by

$$C.\text{Sh} : \mathbf{UF} \quad \text{‘shapes’} \qquad C.\text{Po} : C.\text{Sh} \rightarrow \mathbf{UF} \quad \text{‘positions’}$$

We may also write  $S \triangleleft P$  for the container with shape set  $S$  and position family  $P$ , and sometimes  $(x : S) \triangleleft P[x]$  instead of  $S \triangleleft \lambda x \mapsto P[x]$ .

Crucially, for any such  $C = S \triangleleft P$ , we write  $CX$  for the *extension* of the container, i.e., the type

$$(s : S) \times (p : P s) \sqsupset X$$

which you can think of as a tag chosen from  $S$  paired with a tuple of  $X$ s whose size (and structure) is determined from the tag by  $P$ . Our binary tree generators form a fine example!

---

<sup>1</sup>Some people call these ‘sums’. Other people call these ‘products’. The fact that both of these names are used means neither of them helps.