

# Thud and Blunder

Conor Mc Bride and ...?

February 4, 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

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

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

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

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  $\text{leafy} = \text{mystery } \bullet = \text{mystery } \text{‘thud’}$ . 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 promise you 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!