Clowns to the left of me, jokers to the right

(Dissecting Data Structures)

Conor McBride

October 2, 2006

1 Introduction

Think of binary trees and their fold operator. I shall work in an informal, slightly categorical programming language, but what follows is certainly expressible in Haskell or ML.

```
data Tree = Leaf | Node Tree Tree

fold : T → (T → T → T) → Tree → T
fold l n Leaf         = l
fold l n (Node a b) = n (fold l n a) (fold l n b)
```

Let's make this operation tail-recursive (hence strict). We'll need an auxiliary data structure to remember where we are, a bit like a zipper but lop-sided—the Trees we've already visited have become Ts.

```
fold : T → (T → T → T) → Tree → T
fold l n t = inward t ε where

  inward : Tree → List (T + Tree) → T
  inward Leaf          γ = outward l γ
  inward (Node a b) γ = inward a (InL b; γ)

  outward : T → List (T + Tree) → T
  outward t ε           = t
  outward t (InL b; γ) = inward b (InR t; γ)
  outward t (InR u; γ) = outward (n u t) γ
```

The stack structure List (Tree + T) represents the path of nodes we're currently processing, from the node we're visiting out to the root. For each node in the path, we've either gone left, and have yet to process the Tree on the right, or we've gone right having already collected the T from the left.

This notion of stack is thus a slight generalisation of the basic zipper. If you use this fold to write a Tree endofunction, eg,

```
mirror : Tree → Tree
mirror = fold Leaf (flip Node)
```

the stack type is instantiated to StackTreeTree, an ordinary zipper. If you're Martin Hofmann, David Aspinall or Steffen Jost, and you can find a spare bit in each Node, you might be quite excited by this—the spare bit is enough to code up the left or right choice, so you can construct the stack by consuming the input and construct the output by consuming the stack.

The purpose of this article is to figure out how to do this trick in general, calculating the stack type from the original datatype. Let us begin with simple polynomial data structures, seeking the counterpart to the *derivative*, ∂, which calculates zippers. I call this operator *dissection* and I write it ∇.

Later, I hope to persuade you to extend the idea to dependently typed data structures by working with *indexed containers*. This may seem like a wild leap, but we shall find that the flexibility offered by such a general notion of indexing is just what we need to manage the complexities which would otherwise derail us.

2 Dissecting Polynomials

The polynomial functors are just those generated from the following components:

$$\begin{array}{rr@{\,}l}
\text{constants} & \mathsf{K}\,A\,X &= A \\
\text{identity} & \mathsf{Id}\,X &= X \\
\text{sum} & (F+G)\,X &= (F\,X)+(G\,X) \\
\text{product} & (F\times G)\,X &= (F\,X)\times(G\,X)
\end{array}$$

More generally, I shall presume that all kinds of type constructor are closed pointwise under $\mathsf{0}$, $+$, $\mathsf{1}$ and $\times$.

Simple (inductive) datatypes like Tree can be constructed by taking the (least) fixpoint of a polynomial.

$$\textbf{data}\,\mu F = \mathsf{In}\,(F\,(\mu F))$$

The parameter of the polynomial F is thus instantiated recursively with the datatype being defined. Correspondingly, the uses of Id in F mark the places where recursive subobjects go. We get

$$\begin{array}{l}
\textbf{type}\ \mathsf{TreeF}\ =\ \mathsf{1}+\mathsf{Id}\times\mathsf{Id} \\
\textbf{type}\ \mathsf{Tree}\ =\ \mu\mathsf{TreeF}
\end{array}$$

$$\begin{array}{l@{\,}l}
\mathsf{Leaf} &= \mathsf{In}\,(\mathsf{InL}\ \langle\rangle) \\
\mathsf{Node}\,t\,u &= \mathsf{In}\,(\mathsf{InR}\ \langle t,u\rangle)
\end{array}$$

2.1 Zippers Recapitulated

Just to remind you, the zipper construction calculates a one-hole context in a polynomial by collecting all the ways to replace exactly one of the Ids with a $\mathsf{1}$.

$$\begin{array}{l@{\,}c@{\,}c}
\partial(\mathsf{K}\,A) &= & \mathsf{0} \\
\partial\mathsf{Id} &= & \mathsf{1} \\
\partial(F+G) &= & \partial F+\partial G \\
\partial(F\times G) &= & \partial F\times G+F\times\partial G
\end{array}$$

Now, $\partial F\,X$ represents a one-hole context for an X in an $F\,X$, so we must have an operation which plugs an X into the hole.

$$\begin{array}{rl}
(\lessdot_F) : \partial F\,X \to X \to X & \\
\langle\rangle \lessdot_{\mathsf{Id}} \quad x & = x \\
\mathsf{InL}\,c \lessdot_{F+G}\, x & = \mathsf{InL}\,(c \lessdot_F x) \\
\mathsf{InR}\,c \lessdot_{F+G}\, x & = \mathsf{InR}\,(c \lessdot_G x) \\
\mathsf{InL}\ \langle c,g\rangle \lessdot_{F\times G}\, x & = \langle c \lessdot_F x, g\rangle \\
\mathsf{InR}\ \langle f,c\rangle \lessdot_{F\times G}\, x & = \langle f, c \lessdot_G x\rangle
\end{array}$$

To construct the type of zippers in a μF, we first instantiate X with μF—$\partial F\,(\mu F)$ represents one step on the path from the hole to the root. The zipper type thus takes lists of these steps.

$$\mathbb{A}F\ =\ \mathsf{List}\,(\partial F\,(\mu\,F))$$

For our Tree example,

$$\mathbb{A}\mathsf{TreeF} \;=\; \mathsf{List}\,((0 + (1 \times \mathsf{Id} + \mathsf{Id} \times 1))\;\mathsf{Tree}) \;\cong\; \mathsf{List}\,(\mathsf{Tree} + \mathsf{Tree})$$

We thus get a recursive function which rebuilds a tree from a zipper and a subtree.

$$\begin{array}{l}
(\,\lhd\!\!+\!\!+\,) : \mathbb{A}F \to \mu F \to \mu F \\
\qquad\quad [] \lhd\!\!+\!\!+\, t \;= t \\
(c : cs) \lhd\!\!+\!\!+\, t \;= cs \lhd\!\!+\!\!+\, \mathsf{In}\,(c \lhd\!\!+ t)
\end{array}$$

Polynomials are closed under composition, which I shall denote

$$F[G]\,X = F\,(G\,X)$$

and it is easy to check that

$$\partial(F[G]) \cong (\partial F)[G] \times \partial G$$

2.2 From derivatives to dissection

The dissection construction is similar in character to the derivative, but it transforms a unary functor F into a *binary* functor

$$\nabla F\;C\;J$$

represents an $F\;X$ with a cursor in place of of one X, like the derivative, except that the X's to the left have been replaced by 'clowns' C, and the X's to the right by 'jokers' J. Correspondingly, we get

$$\partial F\;X \;=\; \nabla F\;X\;X$$

We shall need tho simple operators before we can define ∇, one which prefers clowns, the other jokers.

$$\begin{array}{l}
\nwarrow\! F\;C\;J \;= F\;C \\
\nearrow\! F\;C\;J \;= F\;J
\end{array}$$

Now the dissection proceeds exactly as the derivative, except that we recognize left and right appropriately

$$\begin{array}{lcc}
\nabla(\mathsf{K}\,A) & = & 0 \\
\nabla\mathsf{Id} & = & 1 \\
\nabla(F + G) & = & \nabla F + \nabla G \\
\nabla(F \times G) & = & \nabla F \times \nearrow\! G + \nwarrow\! F \times \nabla G
\end{array}$$

Our general stack type is now given just like the zipper

$$\mathsf{Stack}_F\;C\;J = \mathsf{List}\,(\nabla F\;C\;J)$$

In particular, for Trees, we get

$$\nabla\mathsf{TreeF} = 0 + (1 \times \nearrow\!\mathsf{Id} + \nwarrow\!\mathsf{Id} \times 1)$$

and hence

$$\mathsf{Stack}_{\mathsf{TreeF}}\;C\;J = \mathsf{List}\,(0 + (1 \times J + C \times 1)) \cong \mathsf{List}\,(J + C)$$

exactly as we had in the introduction, when the clowns to the left were the computed Ts and jokers to the right were unvisited Trees.

Bifunctors are closed under composition, with

$$F[G, H]\;X\;Y = F\;(G\;X\;Y)\;(H\;X\;Y)$$

and it is easy to check that

$$\nabla(F[G]) \cong (\nabla F)[\nwarrow\! G, \nearrow\! G] \times \nabla G$$

2.3 Getting Around

It's one thing to construct the Stack type, but that still doesn't tell us how to write our tail-recursive fold generically. Given some node, we need to be able to start at the leftmost subnode if there is one, then work our way left-to-right until there are no more. Accordingly, we shall define the following pair of operations

$$\begin{aligned}
\mathsf{into}_F &: \quad F\,J \to J \times \nabla F\,C\,J + F\,C \\
\mathsf{next}_F &: \nabla F\,C\,J \to C \to J \times \nabla F\,C\,J + F\,C
\end{aligned}$$

Note that these operations have the same return type: they each yield either a dissection together with the joker from the cursor position, or a transformed F completely full of clowns. They differ only in their starting point: into starts from an F full of jokers, whilst next starts from an existing dissection, provided you supply a clown to go in the old cursor position.

Notice also that into does not start with any clowns! Correspondingly, it has no way to invent them and can only return an InR if there were no jokers in the input to transform. Similarly, an InL return is bound to leave the cursor in the leftmost position, just because there are no clowns to put to the left of it.

Let us now give the definitions of into and next for polynomials:

$$\begin{aligned}
&\mathsf{into}_{\mathsf{K}A}\ \ a &&= \mathsf{InR}\,a \\
&\mathsf{into}_{\mathsf{Id}}\ \ x &&= \mathsf{InL}\ \langle x, \langle\rangle\rangle \\
&\mathsf{into}_{F+G}\ (\mathsf{InL}\,f) &&= (\mathsf{id} \times \mathsf{InL} + \mathsf{InL})\ (\mathsf{into}_F\ f) \\
&\mathsf{into}_{F+G}\ (\mathsf{InR}\,g) &&= (\mathsf{id} \times \mathsf{InR} + \mathsf{InR})\ (\mathsf{into}_G\ g) \\
&\mathsf{into}_{F\times G}\ \langle f_J, g_J\rangle &&= \mathsf{tryL}\ \langle \mathsf{into}\ f_J, g_J\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathsf{tryL}\ :\ (J \times \nabla F\,C\,J + F\,C) \times G\,J \ \ \to\ \ J \times \nabla(F \times G)\,C\,J + (F \times G)\,C \\
&\mathsf{tryL}\ \langle \mathsf{InL}\ \langle j, f'\rangle\,,\ g_J\rangle = \mathsf{InL}\ \langle j, \mathsf{InL}\ \langle f', g_J\rangle\rangle \\
&\mathsf{tryL}\ \langle \mathsf{InR}\,f_C, \qquad g_J\rangle = \mathsf{tryR}\ \langle f_C, \mathsf{into}_G\ g_J\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathsf{tryR}\ :\ F\,C \times (J \times \nabla G\,C\,J + G\,C) \ \ \to\ \ J \times \nabla(F \times G)\,C\,J + (F \times G)\,C \\
&\mathsf{tryR}\ \langle f_C, \mathsf{InL}\ \langle j, g'\rangle\rangle = \mathsf{InL}\ \langle j, \mathsf{InR}\ \langle f_C, g'\rangle\rangle \\
&\mathsf{tryR}\ \langle f_C, \mathsf{InR}\,g_C\rangle \qquad = \mathsf{InR}\ \langle f_C, g_C\rangle
\end{aligned}$$

As you can see, when finding the leftmost element in a pair, we only try the second component if the first does not deliver a suitable dissection. Note also that a constant is vacuously both 'all jokers' and 'all clowns': $\mathsf{K}\,A\,J = A = \mathsf{K}\,A\,C$.

$$\begin{aligned}
&\mathsf{next}_{\mathsf{K}A}\ \ z && c\ = !\,z \\
&\mathsf{next}_{\mathsf{Id}}\ \ \langle\rangle && c\ = \mathsf{InR}\,c \\
&\mathsf{next}_{F+G}\ (\mathsf{InL}\,f') && c\ = (\mathsf{id} \times \mathsf{InL} + \mathsf{InL})\ (\mathsf{next}_F\ f'\ c) \\
&\mathsf{next}_{F+G}\ (\mathsf{InR}\,g') && c\ = (\mathsf{id} \times \mathsf{InR} + \mathsf{InR})\ (\mathsf{next}_G\ g'\ c) \\
&\mathsf{next}_{F\times G}\ (\mathsf{InL}\ \langle f', g_J\rangle) && c\ = \mathsf{tryL}\ \langle \mathsf{next}_F\ f'\ c, g_J\rangle \\
&\mathsf{next}_{F\times G}\ (\mathsf{InR}\ \langle f_C, g'\rangle) && c\ = \mathsf{tryR}\ \langle f_C, \mathsf{next}_G\ g'\ c\rangle
\end{aligned}$$

These two give us a generic tail-recursive implementation of map, turning jokers into clowns:

$$\begin{aligned}
&\mathsf{map}\ :\ (J \to C) \to F\,J \to F\,C \\
&\mathsf{map}\ \phi\ f\ =\ \mathsf{iter}\ (\mathsf{into}\ f)\ \textbf{where} \\
&\quad \mathsf{iter}\ :\ (J \times \nabla F\,C\,J + F\,C) \to F\,C \\
&\quad \mathsf{iter}\ (\mathsf{InL}\ \langle j, d\rangle) = \mathsf{iter}\ (\mathsf{next}\ d\ (\phi\ j)) \\
&\quad \mathsf{iter}\ (\mathsf{InR}\ f) \qquad = f
\end{aligned}$$

Moreover, we can construct our generic tail-recursive fold operator, where the jokers to the right are unprocessed elements of μF and the clowns to the left are the intermediate values in T awaiting sufficient of their fellows to apply the algebra.

$$\begin{array}{l}
\mathsf{fold} \;:\; (F\,T \to T) \to \mu\,T \to T \\
\mathsf{fold}\;\phi\;r \;=\; \mathsf{inward}\;r\;\varepsilon\;\textbf{where} \\
\quad \mathsf{inward} \;:\; \mu F \to \mathsf{Stack}_F\;T\;(\mu F) \to T \\
\quad \mathsf{inward}\;(\mathsf{In}\;f)\;\gamma \;=\; \mathsf{onward}\;(\mathsf{into}\;f)\;\gamma \\
\quad \mathsf{outward} \;:\; T \to \mathsf{Stack}_F\;T\;(\mu F) \to T \\
\quad \mathsf{outward}\;t\;\varepsilon \qquad = t \\
\quad \mathsf{outward}\;t\;(d;\gamma) = \mathsf{onward}\;(\mathsf{next}\;d\;t)\;\gamma \\
\quad \mathsf{onward} \;:\; (\mu F \times \nabla F\;T\;(\mu F) + F\;T) \to \mathsf{Stack}_F\;T\;(\mu F) \to T \\
\quad \mathsf{onward}\;(\mathsf{InL}\;\langle r, d\rangle)\;\gamma \;= \mathsf{inward}\;r\;(d;\gamma) \\
\quad \mathsf{onward}\;(\mathsf{InR}\;f) \qquad \gamma \;= \mathsf{outward}\;(\phi\;f)\;\gamma
\end{array}$$

2.4 Division is dissection with no clowns

Dissection gives us a ready way to capture the concept of 'far left'.[1] Let us define

$$\ell F = \nabla F\,\mathsf{0}$$

From the laws for ∇, we get laws for ℓ

$$\begin{array}{lcr}
\ell(\mathsf{K}\,A) & = & \mathsf{0} \\
\ell\mathsf{Id} & = & \mathsf{1} \\
\ell(F+G) & = & \ell F + \ell G \\
\ell(F \times G) & = & \ell F \times G \;+ F[\mathsf{0}] \times \ell G
\end{array}$$

Any polynomial container either has a leftmost element or no elements at all. That is, we may establish the *remainder* theorem: for all polynomials F,

$$F \;\cong\; \mathsf{Id} \times \ell F + F[\mathsf{0}]$$

or, more explicitly,

$$F\,X \;\cong\; X \times \ell F\,X + F\,\mathsf{0}$$

hence we may think of ℓ as *division.*

Now, from left to right, our into operation will serve admirably if we take the type of clowns to be $\mathsf{0}$. We shall need to check that we can define its inverse:

$$\begin{array}{l}
\mathsf{lout} \;:\; X \times \ell F\,X + F\,\mathsf{0} \to F\,X \\
\mathsf{lout}\;(\mathsf{InL}\;p) \;\;\, = \mathsf{lplug}\;p \\
\mathsf{lout}\;(\mathsf{InR}\;f_0) = F\;!\;f_0
\end{array}$$

$$\begin{array}{lll}
\mathsf{lplug}_F \;:\; X \times \ell F\,X \to F\,X & & \\
\mathsf{lplug}_{\mathsf{K}A} & \langle x, z\rangle & = !z \\
\mathsf{lplug}_{\mathsf{Id}} & \langle x, \langle\rangle\rangle & = x \\
\mathsf{lplug}_{F+G} & \langle x, \mathsf{InL}\;f'\rangle & = \mathsf{InL}\;(\mathsf{lplug}_F\;x\;f') \\
\mathsf{lplug}_{F+G} & \langle x, \mathsf{InR}\;g'\rangle & = \mathsf{InR}\;(\mathsf{lplug}_G\;x\;g') \\
\mathsf{lplug}_{F\times G} & \langle x, \mathsf{InL}\;\langle f', g\rangle\rangle & = \langle \mathsf{lplug}_F\;x\;f', g\rangle \\
\mathsf{lplug}_{F\times G} & \langle x, \mathsf{InR}\;\langle f_0, g'\rangle\rangle & = \langle F\;!\;f_0, \mathsf{lplug}_G\;x\;g\rangle
\end{array}$$

We may readily show that into and lout are mutual inverses by induction on the structure of polynomials.

Now, if we can play this trick once, we can play it again:

$$F\,X \;\cong\; X \times X \times \ell\ell F\,X + X \times \ell F\,\mathsf{0} + F\,\mathsf{0}$$

[1] The less said about the far right, the better.

Indeed, we should like to hope that iterating the process until all the (finitely many) elements have been found will yield the power series expansion:

$$F\,X \;\cong\; \sum_{n\in\omega} X^n \times \ell^n F\,\mathsf{0}$$

Let's prove this by induction on the structure of polynomials. The cases for $\mathsf{K}\,A$ and Id are trivial. The case for $+$ follows readily because ℓ commutes with $+$. The case for $\times$ is more convoluted. We must show that

$$\left(\sum_{i\in\omega} X^i \times \ell^i F\,\mathsf{0}\right) \times \left(\sum_{j\in\omega} X^j \times \ell^j G\,\mathsf{0}\right) \;\cong\; \sum_{n\in\omega} X^n \times \ell^n (F\times G)\,\mathsf{0}$$

Perhaps the usual trick will work. I suspect Tom Körner's Pavlovian teaching methods will ensure that I can only do this in my sleep. The left-hand side is just a big dependent record, so we may permute the fields as long as we respect dependency:

$$LHS \;\cong\; \sum_{i\in\omega}\sum_{j\in\omega} X^{i+j} \times \ell^i F\,\mathsf{0} \times \ell^j G\,\mathsf{0}$$

Next we change variables, abstracting $n = i + j$,

$$\cdots \;\cong\; \sum_{n\in\omega} X^n \times \sum_{i\leq n} \ell^i F\,\mathsf{0} \times \ell^{n-i} G\,\mathsf{0}$$

And this will certainly equal the right-hand side if we can show the highly plausible

$$\ell^n (F\times G)\,\mathsf{0} \;\cong\; \sum_{i\leq n} \ell^i F\,\mathsf{0} \times \ell^{n-i} G\,\mathsf{0}$$

asserting that any n-element $F \times G$ is made from an i-element F and an $(n-i)$-element G. This should go by induction on n, for arbitrary polynomials F and G. The base case indeed asserts

$$(F\times G) \;\cong\; \sum_{i\leq 0} \ell^i F\,\mathsf{0} \times \ell^{n-i} G\,\mathsf{0} \;\cong\; \ell^0 F\,\mathsf{0} \times \ell^0 G\,\mathsf{0} \;\cong\; F\,\mathsf{0} \times G\,\mathsf{0}$$

In the step case, it's tempting to split ℓ^{n+1} as $\ell\ell^n$, but the bracketing won't let us apply the inductive hypothesis. Go the other way.

$$\begin{aligned}
\ell^{n+1}(F\times G)\,\mathsf{0} &\cong\; \ell^n \ell(F\times G)\,\mathsf{0}\\
&\cong\; \ell^n(\ell F \times G + F[\mathsf{0}] \times \ell G))\,\mathsf{0}\\
&\cong\; \ell^n(\ell F \times G)\,\mathsf{0} + \ell^n(F[\mathsf{0}] \times \ell G))\,\mathsf{0}\\
&\cong\; \ell^n(\ell F \times G)\,\mathsf{0} + F\,\mathsf{0} \times \ell^n \ell G\,\mathsf{0}\\
&\cong\; \left(\textstyle\sum_{i\leq n} \ell^i \ell F\,\mathsf{0} \times \ell^{n-i} G\,\mathsf{0}\right) + F\,\mathsf{0} \times \ell^n \ell G\,\mathsf{0}\\
&\cong\; F\,\mathsf{0} \times \ell^{n+1} G\,\mathsf{0} + \textstyle\sum_{i\leq n} \ell^{i+1} F\,\mathsf{0} \times \ell^{(n+1)-(i+1)} G\,\mathsf{0}\\
&\cong\; \textstyle\sum_{i\leq n+1} \ell^i F\,\mathsf{0} \times \ell^{n+1-i} G\,\mathsf{0}
\end{aligned}$$

Remark. Experience from calculus tempts one to seek a more general result, viz.

$$\ell^n (F\times G) \;\cong\; \sum_{i\leq n} \ell^i F \times \ell^{n-i} G \qquad (\times)$$

which surely holds if ℓ is replaced by ∂—'find n holes in a pair by finding i in one and the rest in the other'—but ℓ finds *leftmost* holes, so it's not hard to see why the above is bogus.

2.5 Taylor's theorem without quotients

In this section, let's establish combinatorial connections between ℓ and ∂. The former finds holes left-to-right, the latter in any order, so we should expect some sort of permutative relationship between them. Let's start with some basics.

Thinking geometrically, if there's only one hole, then any hole is leftmost:

$$\ell F[\mathsf{0}] \;\cong\; \partial F[\mathsf{0}] \qquad \text{ie} \qquad \ell F\,\mathsf{0} \;\cong\; \partial F\,\mathsf{0}$$

Let's check this inductively. The only interesting case is that of pairs.

$$\begin{array}{l} \ell(F\times G)\,\mathsf{0} \;=\; (\ell F\times G + F[\mathsf{0}]\times \ell G)\,\mathsf{0} \;=\; \ell F\,\mathsf{0}\times G\,\mathsf{0} + F\,\mathsf{0}\times \ell G\,\mathsf{0} \\ \partial(F\times G)\,\mathsf{0} \;=\; (\partial F\times G + F\times \partial G)\,\mathsf{0} \;=\; \partial F\,\mathsf{0}\times G\,\mathsf{0} + F\,\mathsf{0}\times \partial G\,\mathsf{0} \end{array}$$

so the inductive hypotheses pull us through.

What if there's more than one hole? Again, thinking geometrically, one would tend to hope that

$$\ell\partial = \ell\ell + \partial\ell$$

Picking 'any hole then the leftmost remaining' can happen in two ways: the first hole picked happens to be the leftmost and the second is the next leftmost, or the first hole picked is not leftmost so the second is. It's not so hard to check this by polynomial induction: as two pickings yields $\mathsf{0}$ for constants and Id, and ℓ and ∂ both commute with $+$, the only interest lies amongst the pairs. Expanding the definitions and applying the inductive hypotheses just serves to enumerate the possibilities by brute force: 'leftmost and another on the left', 'the only one on the left, one on the right', or 'none on the left, so leftmost and another on the right'.

$$\begin{array}{l} \ell\partial(F\times G) \\ \qquad\cong\; \ell(\partial F\times G + F\times\partial G) \;\cong\; \ell(\partial F\times G) + \ell(F\times\partial G) \\ \qquad\cong\; \ell\partial F\times G + \partial F[\mathsf{0}]\times \ell G + \ell F\times\partial G + F[\mathsf{0}]\times \ell\partial G \\ \qquad\cong\; (\ell\ell F + \partial\ell F)\times G + \partial F[\mathsf{0}]\times\ell G + \ell F\times\partial G + F[\mathsf{0}]\times(\ell\ell G + \partial\ell G) \\ (\ell\ell + \partial\ell)(F\times G) \\ \qquad\cong\; \ell\ell(F\times G) + \partial\ell(F\times G) \\ \qquad\cong\; \ell(\ell F\times G + F[\mathsf{0}]\times\ell G) + \partial(\ell F\times G + F[\mathsf{0}]\times\ell G) \\ \qquad\cong\; \ell(\ell F\times G) + F[\mathsf{0}]\times\ell\ell G + \partial(\ell F\times G) + F[\mathsf{0}]\times\partial\ell G \\ \qquad\cong\; \ell\ell F\times G + \ell F[\mathsf{0}]\times\ell G + F[\mathsf{0}]\times\ell\ell G + \partial\ell F\times G + \ell F\times\partial G + F[\mathsf{0}]\times\partial\ell G \\ \qquad\cong\; (\ell\ell F + \partial\ell F)\times G + \ell F[\mathsf{0}]\times\ell G + \ell F\times\partial G + F[\mathsf{0}]\times(\ell\ell G + \partial\ell G) \end{array}$$

By the above result, we have that these are equal.

Iterating this, we learn that

$$\ell^n\partial = \partial\ell^n + n\times\ell^{n+1}$$

which is also combinatorially intuitive—there are n ways the picking strategy on the left can select the leftmost $n+1$ elements.

Now let's go for the main result—finding all n holes left-to-right then permuting them is the same as finding them in any order.

$$\partial^n F\,\mathsf{0} \cong n!\times\ell^n F\,\mathsf{0}$$

We proceed by induction on n with arbitrary F. The base case is trivial. For the step, we have

$$\begin{array}{l}
\partial^{n+1} F\,\mathsf{0} \\
\quad\cong\ \{\text{decompose } \partial^{n+1}\} \\
\partial^{n} \partial F\,\mathsf{0} \\
\quad\cong\ \{\text{inductive hypothesis for } \partial F\} \\
n! \times \ell^{n} \partial F\,\mathsf{0} \\
\quad\cong\ \{\partial \text{ through } \ell^{n}\} \\
n! \times (\partial \ell^{n} + n \times \ell^{n+1})\, F\,\mathsf{0} \\
\quad\cong\ \{\text{distribute}\} \\
n! \times \partial \ell^{n} F\,\mathsf{0} + n \times n! \times \ell^{n+1} F\,\mathsf{0} \\
\quad\cong\ \{\text{any unique is leftmost for } \ell^{n} F\} \\
n! \times \ell \ell^{n} F\,\mathsf{0} + n \times n! \times \ell^{n+1} F\,\mathsf{0} \\
\quad\cong\ \{\text{distribute}\} \\
(n+1) \times n! \times \ell^{n+1} F\,\mathsf{0} \\
\quad\cong\ \{\text{definition of factorial}\} \\
(n+1)! \times \ell^{n+1} F\,\mathsf{0}
\end{array}$$