Differential Calculus for Datatypes

Flyxion
Independent Researcher

November 2026

Abstract

Differentiating a datatype describes the information needed to replace a selected occurrence while preserving its surroundings. The argument begins with a pair, derives the representation for a list, and uses containers to explain why the same construction works across a broad class of datatypes. A final development applies the zipper to expression syntax with variable binding, where reconstruction, scope, and typing are shown to be three separate layers of information. The mathematical construction is kept distinct from the editing operations it makes possible, and no appeal is made to rates of change or limits.

Summary of Results and Scope

Throughout, $\cong$ denotes isomorphism of types, a prime denotes the formal derivative with respect to X, ∂_Z denotes the formal derivative with respect to a named variable Z, $\mathrm{List}(A)$ denotes the type of finite sequences of A, and $L(X)$ is the type of finite lists with elements of type X. The essay establishes the following, each with a derivation in the body. Differentiation of sums, products, and compositions obeys

$$(F+G)' \cong F' + G', \qquad (F \times G)' \cong F' \times G + F \times G', \qquad (F \circ G)' \cong F'(G) \times G'.$$

For a least fixed point μF of a polynomial expression $F(X, Z)$, the derivative is a local factor paired with a stack of steps,

$$\partial_X(\mu F) \cong (\partial_X F)[\mu F/Z] \times \mathrm{List}((\partial_Z F)[\mu F/Z]).$$

For lists this gives $L' \cong L \times L$, and repeated differentiation gives $L^{(k)} \cong k!\, L^{k+1}$. For expression syntax with variable binding, the essay separates reconstruction, scope, and typing as three layers of information, and illustrates the separation with the replacement of the domain of a dependent function type. That separation is the organizing claim of the essay and is supported by a worked example, whereas the formulas above are derived.

The essay is an exposition, and its results are not new. The identification of the derivative of a regular type with its type of one-hole contexts is due to McBride, its categorical account to Abbott, Altenkirch, Ghani, and McBride, its implementation as a program on datatype codes to Morris, Altenkirch, and McBride, and the zipper to Huet. The contribution lies in the

arrangement of the derivations and in the discussion of binding. The essay does not survey the literature on structural editors, and it does not claim that contexts are the only possible account of selection, only that the construction given here produces them from a specification of the datatype. Only algebraic notions are used, and no appeal is made to rates of change or limits. A section on specifications gives a grammar for datatype layers and a Lean 4 implementation of the derivative, the plugging operation, and the enumeration of contexts, tested on lists and binary trees but not proved correct in general.

1 Selection and Reconstruction

Consider a concrete programming problem. Given the list $[a, b, c]$, select b and retain enough information to replace it later. Removing b gives $[a, c]$ and leaves its original position undetermined, because a replacement value could be placed before, between, or after the survivors. The prefix $[a]$ and the suffix $[c]$, retained separately, supply the missing information. They can be reused to insert b, restoring the original list, or to insert d, yielding $[a, d, c]$.

The data retained around a selected position, together with whatever is required for reconstruction, will be called a context. The general problem is then to determine whether, given a specification of a datatype, a specification of its contexts can be derived systematically. The remainder of the essay argues that it can, and that the derivation coincides with formal differentiation of the specification, a correspondence developed by McBride [1] and given a categorical account in the work on containers [7, 5, 6]. The zipper of Huet [4] is the practical data structure that this derivation explains.

2 The Algebra of Datatype Specifications

The derivation needs only a small fragment of type algebra. The type 0 has no inhabitants and 1 has a single inhabitant. The type $F + G$ represents a choice between an F and a G, and $F \times G$ represents a pair. The variable X marks the occurrences eligible for selection. A constant type can carry data while contributing no positions of type X to this operation, so a constant is invisible to selection even though it is stored.

The type $X^2 = X \times X$ gives the first example. Its derivative is $X + X$. Either summand stores one surviving value, and the choice of summand identifies which slot was selected. The type $1 + X$ shows what the constants mean. The empty case has no position to select, so it contributes nothing to the derivative, while the occupied case has exactly one position. Once that position is selected nothing further needs to be stored, so the context type is 1, and $(1 + X)' \cong 1$.

3 The Sum and Product Rules as Construction Rules

The rules of differentiation can be stated and then read as procedures for constructing a context:

$$(F + G)' \cong F' + G', \qquad (F \times G)' \cong (F' \times G) + (F \times G').$$

For a sum, the original alternative determines which kind of context is possible, since a value built with the left injection can only have a hole inside an F. For a product, selection takes place in one factor and the other factor survives intact. This gives a reason for each summand of the product rule. A term of $F' \times G$ remembers a context in F and the whole value of G, and the other summand exchanges their roles.

Reconstruction follows the same shape. The tag of the sum tells the reconstruction operation which factor receives the replacement, and the intact factor supplies the rest of the product. The familiar differentiation rule therefore begins to specify what a program must do, which is the sense in which the calculus is a calculus of programs and not only an analogy with analysis.

4 The Chain Rule for Nested Datatypes

Composition of datatypes has its own rule. Let $F(G(X))$ be a structure whose outer positions each contain a $G(X)$. Selecting an X requires two choices. First the outer position containing the relevant $G(X)$ is selected, leaving a context of type $F'(G(X))$. Then an X is selected inside that inner structure, leaving a context of type $G'(X)$. Thus

$$(F \circ G)'(X) \cong F'(G(X)) \times G'(X).$$

In the container formulation developed below, a shape of $F \circ G$ is a shape of F with a shape of G at each of its positions, and a position is a pair of an outer position and an inner position, so that removing one pair leaves the other outer entries whole and the inner structure at the chosen outer position with one hole. Reconstruction runs in the reverse order, since the inner context is filled with the replacement X first and the outer context is then filled with the rebuilt $G(X)$.

5 The Derivative of the List Type

The main worked derivation starts from the recursive description of finite lists,

$$L(X) \cong 1 + X \times L(X).$$

Differentiating with respect to the element type X gives

$$L'(X) \cong L(X) + X \times L'(X).$$

The two cases are read in order. If the selected position is the head, the retained data are the tail. If the selected position lies farther along, the head is retained and the selection continues in the tail. Because the original list is finite, the second case can occur only finitely many times before the first case supplies the suffix. The retained heads form the prefix, which yields

$$L'(X) \cong L(X) \times L(X).$$

The closed form can be checked directly. The type $L \times L$ satisfies the same equation, since

$L \times L \cong (1 + X \times L) \times L \cong L + X \times (L \times L)$. The finite contexts generated by the two cases of the equation are isomorphic to $L \times L$, as the following argument shows in both directions. A recursive context yields a prefix by collecting each retained head, with the outermost layer holding the first element of the list, so that the prefix comes out in list order; the zipper stack of a later section lists steps from the hole outward and therefore holds the same heads in reverse order. The suffix comes from the terminating case. Conversely, a prefix and a suffix determine the recursive context by wrapping the terminating case once for each prefix element. The opening example is thereby connected to a derivation whose steps account for every component of the representation, so the pair of prefix and suffix is no longer a representation that merely happened to work.

With $L' \cong L^2$ derived, the chain rule of the preceding section applies to a list of pairs, taking $F = L$ and $G(X) = X^2$, and gives

$$\big(L(X^2)\big)' \cong L(X^2) \times L(X^2) \times (X + X).$$

The two lists retain the complete pairs before and after the selected pair, and the final sum retains the other member of that pair and records which slot was selected. This is a small case in which the chain rule specifies two successive reconstruction operations.

6 Branching Datatypes and the Path to a Position

The argument extends to a binary tree whose leaves carry values of type X,

$$T(X) \cong X + T(X) \times T(X).$$

Its derivative satisfies

$$T'(X) \cong 1 + \big(T'(X) \times T(X)\big) + \big(T(X) \times T'(X)\big).$$

The unit case marks arrival at the selected leaf. Each of the other cases retains the direction taken and the entire sibling subtree. A tree context therefore records a path together with the data needed to rebuild the tree along that path. The distinction matters because two trees can share the same route to a leaf while differing in every subtree beside it. The route specifies where to go, and the context also supplies what to put back around the replacement.

7 Fixed Points and the Zipper Stack

The list and tree derivations were obtained by differentiating a recursive equation and then reading the result, a step that leaves open how a general procedure would reach the answer. Morris, Altenkirch, and McBride supply that procedure by implementing differentiation in Epigram as an ordinary program on codes for datatypes [2]. The program takes a type expression and a variable to differentiate with respect to, and returns a new type expression describing the contexts. It handles the empty and unit types, sums, and products by the rules of the

previous sections, and it also handles least fixed points, which is the case of interest for recursive datatypes. The treatment of Hinze, Jeuring, and Löh in Generic Haskell, which Morris and colleagues cite, covers polynomials in one variable and has no fixed-point case; access to the full syntax of type expressions is what permits several parameters and hence fixed points.

Write a recursive type as μF, where $F(X, Z)$ is a polynomial expression with the element variable X and a variable Z marking the recursive children. A selected X lies at the end of a path through the recursive structure, and the following rule records the two kinds of content along that path.

Proposition 1 (Fixed-point rule)**.** *Let $F(X, Z)$ be built from constants, X, Z, sums, and products, and let μF be its least fixed point in Z. Then*

$$\partial_X(\mu F) \;\cong\; (\partial_X F)[\mu F/Z] \;\times\; \mathrm{List}((\partial_Z F)[\mu F/Z]).$$

The proposition is established by the derivation below. The first factor is the context within the node where selection ends, a single layer in which the X is chosen directly and every recursive child is left as a whole subtree. The second factor is the zipper stack, a list of steps each giving the context for a Z inside an F, which rebuilds the structure from that node back to the root.

The formula follows from the chain rule in two variables. The earlier rule treats $F(G(X))$, whereas here $F(X, R(X))$ depends on X both directly and through $R(X) = \mu Z.F(X, Z)$, and the derivative has a term for each:

$$\frac{d}{dX}F(X, R(X)) \cong (\partial_X F)(X, R(X)) + (\partial_Z F)(X, R(X)) \times R'(X).$$

Since $R(X) \cong F(X, R(X))$, this is an equation for $R'(X)$, and abbreviating the two layer derivatives as follows:

$$A(X) = (\partial_X F)(X, R(X)), \qquad B(X) = (\partial_Z F)(X, R(X)).$$

A selected occurrence is either an X directly in the current layer, contributing $A(X)$, or an X inside a recursive child. The latter case contributes a layer context $B(X)$ together with a context $R'(X)$ inside the child, and hence

$$R'(X) \cong A(X) + B(X) \times R'(X).$$

The finite contexts generated by this equation, those in the initial algebra of $C \mapsto A + B \times C$, consist of a finite sequence of $B(X)$ steps ending in one $A(X)$ context, so that

$$R'(X) \cong \sum_{n \geq 0} B(X)^n \times A(X) \cong A(X) \times \mathrm{List}\big(B(X)\big).$$

The recursive equation reads the steps from the root toward the selected node, whereas the zipper stack reads from the selected node toward the root, so under reversal the $B(X)^n \times A(X)$ of the equation becomes the local context $A(X)$ at the hole followed by n steps leading outward

to the root. The derivation also shows exactly where the least fixed point matters, which is that every context has a finite path ending in a local selection.

The two earlier examples fall out directly. For lists, $F(X, Z) = 1 + X \times Z$, so $\partial_X F = Z$ and $\partial_Z F = X$. Substituting $\mu F = L(X)$ gives $\partial_X L \cong L(X) \times \mathrm{List}(X) \cong L(X) \times L(X)$. The first factor is the tail retained when the head is selected, and each step in the stack is a retained head, which is the prefix in reverse order. For binary trees, $F(X, Z) = X + Z \times Z$ gives $\partial_X F = 1$ and $\partial_Z F = Z + Z$, so $T'(X) \cong \mathrm{List}\big(T(X) + T(X)\big)$. Each step is a choice of direction together with the sibling subtree, and the unit factor marks arrival at the leaf. This agrees with the recursive equation of the previous section, since solving $T' = 1 + 2\,T\,T'$ gives the same list of direction-and-sibling pairs. The same rule applies when constructors have different numbers of recursive children. Let $U(X) = \mu Z.(X + Z^2 + Z^3)$ be a tree with binary and ternary nodes, so that $F(X, Z) = X + Z^2 + Z^3$, $\partial_X F = 1$, and $\partial_Z F = 2Z + 3Z^2$. A step through a binary node records which of its two children was selected and retains one sibling as a whole subtree, while a step through a ternary node records which of its three children was selected and retains two siblings. The substitution $[\mu F/Z]$ is applied after differentiation, so the summands of $\partial_Z F$ record the choices available within a single layer, and the result is

$$U'(X) \cong \mathrm{List}\big(2U(X) + 3U(X)^2\big).$$

The local factor never needs more than the single layer in which the X sits. The hand calculation and the generic rule therefore coincide, and the rule explains why the stack, and not a single layer, is the right shape for a context in a recursive type.

A note on implementation is useful here and is kept short. The pair of operations that plug a value back into a node context and plug a rebuilt subtree back into the zipper stack are mutually recursive. The Epigram implementation avoids a mutual definition by defining a family of plugging tasks and interpreting it by recursion on the context [2]. The structurally recursive pattern matching in such definitions is translated into a traditional type theory with universes, inductive types and relations, and the axiom K, by Goguen, McBride, and McKinna. That translation preserves reduction and supplies a termination argument for definitions of this kind [3]. That connection lets a reader follow the program beneath the notation without interrupting the derivation.

8 Containers and the General Construction

The preceding derivations are anchored in McBride's regular-type manuscript [1]. The container formulation developed by Abbott, Altenkirch, Ghani, and McBride [5, 6] gives a complementary account of the same result. Where the Epigram program computes the type of a context from the syntax of a declaration, containers describe what a context contains in terms of shapes and positions. A container describes a set of shapes S and, for each shape s, a set of positions $P(s)$. A value of the container consists of a shape and an assignment of values to its positions [7].

For a container with finitely many positions in each shape, the derivative can be written

$$F'(X) \;\cong\; \sum_{s \in S} \sum_{p \in P(s)} X^{P(s)\setminus\{p\}},$$

where X^A means an assignment of a value of type X to each position in A. The derivative remembers the original shape, the selected position, and the values at all remaining positions.

For a list, the shape is its length n and the positions are the indices $0, \ldots, n-1$. Choosing an index i and keeping the other values, then splitting those values at i, recovers the prefix and suffix representation. The formula here is applied to the whole recursive structure, whose shapes already include the length of the list, and not to a single layer F. It is therefore a different presentation of the construction of the previous section and does not replace it: the fixed-point rule computes the same type layer by layer, as a local factor and a stack, while the container formula describes it in one step from the shapes and positions of the finished structure. This gives two views of the same object, one derived recursively from the defining equation and one described directly by positions. Altenkirch and McBride show how descriptions of this kind can be written inside a dependently typed language and used to define generic operations [8], which is the setting in which the derivative becomes an ordinary program and not only a piece of mathematics.

9 From Contexts to Zippers

A context should be distinguished from a focused value. A context of type $F'(X)$ has a position ready to receive an X. Pairing it with an X gives a focused structure, and reconstruction is expressed as a map

$$\mathrm{plug}\colon X \times F'(X) \longrightarrow F(X).$$

The map plug forgets the focus. The exact correspondence is between a context paired with a value and a structure paired with a valid position. In the container description, with shapes S and positions $P(s)$,

$$X \times F'(X) \;\cong\; \sum_{s \in S} (P(s) \times X^{P(s)}),$$

because $X \times X^{P(s)\setminus\{p\}} \cong X^{P(s)}$ for each selected position p. The forward direction plugs the value into the selected position, which gives a full assignment of values, and retains the position, which gives a focused structure. The inverse removes the value at the retained position and returns it alongside the resulting context. The map plug is the composition of this isomorphism with the map that forgets the position. A zipper move then has a precise description as a change of the distinguished position that preserves the underlying structure.

A list zipper moves its focus by transferring an element between the prefix and the suffix while preserving the order of the reconstructed list. With the prefix stored in reverse order, a move to the right takes a zipper with reversed prefix p, focus x, and suffix $y{:}s$ to one with reversed prefix $x{:}p$, focus y, and suffix s: the old focus is pushed onto the reversed prefix, and the head of the suffix becomes the new focus. A move to the left is the mirror image, and each

move changes only the heads of the relevant lists. The derivative describes the information the representation must preserve, and the representation itself can be chosen to make particular movements efficient. This is the relation between the mathematics and Huet's zipper [4]: the zipper is an inhabitant of $X \times F'(X)$, encoded so that local moves are cheap.

10 Specifications and a Lean Implementation

The derivations above start from a specification written as a type equation, and the same specification can be written as a small grammar and processed by a program. For a datatype with one element parameter and one recursive type variable, a specification is a recursive type $\mu Z.F(X, Z)$, where the layer F follows the grammar

$$F \ ::= \ 0 \ \mid \ 1 \ \mid \ X \ \mid \ Z \ \mid \ F + F \ \mid \ F \times F,$$

in which X marks an element position and Z marks a recursive child. Converting a datatype declaration into this form takes four steps. The alternatives of the declaration become the summands of a sum, the fields of each constructor become the factors of a product, a field holding an element becomes X, and a field holding a recursive occurrence of the type itself becomes Z. A constructor with no fields contributes 1. A list with constructors for the empty list and for a head and tail becomes $1 + X \times Z$, and a binary tree with a leaf carrying an element and a node with two subtrees becomes $X + Z \times Z$. This grammar leaves out arbitrary constant field types beyond 0 and 1, which would be leaves contributing no positions, nested fixed points, and several parameters. Morris, Altenkirch, and McBride handle those by indexing the codes with a number of variables and adding a local-definition form [2], and the present listing keeps the smaller grammar so that the correspondence with the essay stays visible.

The Lean 4 development in the appendix represents layers as a datatype of codes, interprets a code as the elements of one layer with a type of element values and a type of recursive children, takes the least fixed point by nesting that interpretation inside itself, and defines the two layer derivatives by the rules of the earlier sections. The context type for a recursive type is the pair in the fixed-point rule: a layer context for the node where selection ends, and a list of layer contexts for recursive steps, ordered from the hole outward. The function `plug` rebuilds a structure from a context and a replacement, folding the steps over the result of filling the local context, and `holes` enumerates every context of a structure together with the selected element.

The full listing is given in the appendix and is also supplied as a separate file. Its correspondence with the fixed-point rule is direct. The field `local_` of `Ctx` has the type $(\partial_X F)[\mu F/Z]$, the terminal layer context. The field `steps` has the type $\mathrm{List}((\partial_Z F)[\mu F/Z])$, and it stores the path in reconstruction order, from the hole outward. In `plug`, the call to `foldl` starts from the structure rebuilt at the terminal layer and plugs it into each successive parent in turn.

Evaluating the final lines of the listing under Lean 4 gives the following results. For the list $[1, 2, 3]$ the selection depths, meaning the lengths of the zipper stack, are 0, 1, and 2 for the three elements, which is the length of the reversed prefix. Replacing each selected element by 9 gives $[9, 2, 3]$, $[1, 9, 3]$, and $[1, 2, 9]$, and replacing each element by itself restores the list. For the tree with leaves 1, 2, 3 in which the first two share a parent, the depths are 2, 2, and 1.

Replacing each selected leaf by 9 gives the trees printed as ((9 2) 3), ((1 9) 3), and ((1 2) 9), and replacing each leaf by itself restores the printed form ((1 2) 3) of the whole tree. A test on the leaf sequence alone would not detect a change in the shape of the tree, which is why the tree test compares a constructor-preserving view. These are tests on two examples. The essay does not prove in general that `plug` applied to a context from `holes` restores the original, and `holes` is declared `partial` because Lean does not recognize the recursion through the nested type as structural, although it terminates on every finite value.

11 Expressions with Binding

A tree zipper stores the direction taken and the sibling subtree at each step. Syntax with variable binding raises an additional question, because the focus is also situated in a scope, and the variables available there depend on the path taken. A locally nameless representation of the kind described by McBride and McKinna [9] makes the point concrete, although the skeleton below deliberately sets aside the locally nameless discipline in order to isolate the structural derivative. Expressions consist of free variables, bound variables, applications, and dependent function types written $(x{:}A) \to B$, where the body B is a scope in which x is bound.

Ignoring for the moment the distinction between kinds of variable, a skeleton of such syntax has a constant part V for variables, an application with two subexpressions, and a function type with a domain and a body. The context for a single marked subexpression is obtained by introducing a parameter X that permits a marker in place of an entire expression,

$$E_X = \mu Z.(X + V + (Z \times Z)_{\mathrm{app}} + (Z \times Z)_{\Pi}),$$

where $E_X = \mu Z.F(X, Z)$ with $F(X, Z) = X + V + Z^2 + Z^2$, the two copies of $Z \times Z$ being tagged as the application case and the function type case, and then taking

$$C = \partial_X E_X|_{X=0}\,.$$

The derivative selects one marker, and evaluation at $X = 0$ rules out any further markers elsewhere, so that every retained sibling is an ordinary expression $E = E_0$. This is an application of the fixed-point rule of the previous sections to this F. The local factor is $\partial_X F = 1$, which records arrival at the marker. Each step of the stack comes from $\partial_Z F = Z + Z + Z + Z$, one summand for each of the left and right arguments of an application and the domain and body of a function type, and each step retains the sibling expression. The result is $C \cong 1 \times \mathrm{List}(E + E + E + E)$, which satisfies the recursive equation

$$C \cong 1 + (C \times E) + (E \times C) + (C \times E) + (E \times C)$$

obtained by differentiating directly, with the four recursive summands standing for the application left, application right, domain, and body cases in that order; the two copies of each type are distinct cases and are not merged. The empty stack corresponds to a marker that is the whole expression. As a type this is the tree derivative with a larger set of tags, and what the tags mean for editing, namely what a program is entitled to do at the hole, is the subject of

the next paragraphs.

When editing the domain of $(x{:}A) \to B$, the variable x is unavailable, since the domain is checked before the binder exists. When editing the body, x is available and has type A. Both moves descend into a subexpression, and both retain the sibling, but the retained sibling plays different roles. In the second case the retained domain A is the type that the binder gives to the variable, so the sibling the derivative already stores is the material from which the local context $\Gamma,\ x{:}A$ for the hole can be built. McBride and McKinna build this into their zipper (Section 3.1 of [9]): the step for a function-type domain retains the scope, while the step for the body retains a binding, a name paired with the domain, so that the structural context is also a linguistic one. The plain skeleton above retains only the domain, and the binder-aware zipper additionally records the name used when the scope was opened. A context entry therefore does two jobs. It records what is needed to rebuild the whole, and for a binder it records what is needed to determine which variables are available at the focus, since a replacement for the body may mention x and a replacement for the domain may not.

Three layers of information should be kept apart. The derivative supplies the data for reconstruction, and it is a type of one-hole contexts for a datatype that happens to contain binders, with no knowledge of scope. A binder-aware representation, together with an ambient context Γ, a list of name and type pairs, turns the retained entries into an account of which variables are in scope at the focus. Neither of these establishes that a replacement is well typed, which requires checking the replacement against the typing constraints at that position and rechecking any retained parts whose typing depends on it. The distinction matters because a replacement can leave the expression structurally intact while invalidating what was retained. Replacing A in $(x{:}A) \to B$ is always possible structurally, since the body B is kept as it was, yet B was checked under the assumption $x{:}A$. A new domain A' can itself be a valid type while the retained body B fails to typecheck under $x{:}A'$. Rebuilding is therefore a matter of the derivative, scope is a matter of the binder-aware zipper, and fit is a matter of the typing judgment.

12 Iterated Selection and the Limits of the Analogy

The type X^3 provides a compact example of repeated differentiation. The first derivative chooses one of three positions and retains the other two values, so $(X^3)' \cong 3X^2$. The second chooses one of the two remaining positions and retains one value, giving $(X^3)'' \cong 6X$. The coefficient 6 records the six ordered choices of two distinct positions. If the order of selection is immaterial, the corresponding description has three choices, one for each pair of positions. This distinction is the reason to care about the coefficient beyond the arithmetic, because it separates an ordered choice of distinct positions from an unordered set of simultaneously selected positions. Repeated differentiation always selects distinct positions in order, which is why the second derivative of X^3 is $6X$ and not $3X$, since the second selection knows that the first has happened and must avoid it, whereas zipper navigation follows an editing path and can revisit a position.

Lists show what remains between the choices. Starting from $L' \cong L^2$, the product rule gives

$$L'' \cong L'L + LL' \cong 2L^3.$$

The three lists are the segments before the leftmost selected position, between the two selected positions, and after the rightmost selected position. The factor of two records whether differentiation selected the left position first or the right position first, and the middle list can be empty, which is the case of adjacent selected positions. More generally $L^{(k)} \cong k!\, L^{k+1}$. If the first k selections leave $k+1$ segments, the next selection picks one of those segments and splits it in two, and algebraically

$$L^{(k+1)} \cong \left(k!\, L^{k+1}\right)' \cong k!\,(k+1)\, L^k L' \cong (k+1)!\, L^{k+2}.$$

The $k+1$ surviving segments are stored in spatial order, and the $k!$ tags record the permutation in which their separating positions were selected. Given segments $s_0, \ldots, s_k$ and replacement values $x_1, \ldots, x_k$ in left-to-right order, the list is reconstructed as

$$s_0 +\!\!+ [x_1] +\!\!+ s_1 +\!\!+ \cdots +\!\!+ [x_k] +\!\!+ s_k,$$

where $+\!\!+$ denotes concatenation. For example, selecting b and d in $[a, b, c, d, e]$ leaves the segments $[a]$, $[c]$, and $[e]$, and the replacements x_1, x_2 give $[a] +\!\!+ [x_1] +\!\!+ [c] +\!\!+ [x_2] +\!\!+ [e]$. If the replacements arrive in selection order, the permutation is needed to route them to their positions. If they arrive in spatial order, the segments already determine the reconstruction and the permutation tags can be discarded.

The mathematical scope of the essay is limited accordingly. The examples use formal differentiation of datatype descriptions built from sums, products, and finite recursive structures. The argument proceeds through constructions and isomorphisms of types. It can therefore explain what the derivative represents without invoking rates of change or limits, and the word derivative is justified by the algebraic rules it satisfies and not by any analytic content.

13 Conclusion

Returning to $[a, b, c]$, the derivation has added an explanation to the initial observation. The prefix and suffix representation, which at first appeared to be a convenient choice, can now be recovered from the equation defining lists. The corresponding representation for a branching tree follows by the same rules, and both are instances of selection in a container. The practical consequence is that once a datatype specifies its shapes and positions, its derivative states what information an editor must retain to replace a value and reconstruct the whole. For syntax with binding, the account has three layers. The derivative supplies the data for reconstruction, the binder-aware zipper tracks which variables are in scope at the focus, and the typing judgment governs whether a replacement fits.

A Lean Listing

The listing and its output were checked with Lean 4.35.0.

```
set_option linter.unusedVariables false

/-- Layer codes F(X,Z): 0, 1, X, Z, sums and products. -/
inductive Poly : Type
  | zero | one | x | z
  | sum  (p q : Poly)
  | prod (p q : Poly)

open Poly

/-- Elements of one layer, with X-values and Z-values (recursive children). -/
inductive Elt (X Z : Type) : Poly  Type
  | one  : Elt X Z one
  | x    : X  Elt X Z x
  | z    : Z  Elt X Z z
  | inl  {p q : Poly} : Elt X Z p  Elt X Z (sum p q)
  | inr  {p q : Poly} : Elt X Z q  Elt X Z (sum p q)
  | pair {p q : Poly} : Elt X Z p  Elt X Z q  Elt X Z (prod p q)

/-- The least fixed point of a layer. -/
inductive Mu (p0 : Poly) (X : Type) : Type
  | mk : Elt X (Mu p0 X) p0  Mu p0 X

/-- Derivative of a layer with respect to X. -/
def dX : Poly  Poly
  | zero => zero
  | one => zero
  | x => one
  | z => zero
  | sum p q => sum (dX p) (dX q)
  | prod p q => sum (prod (dX p) q) (prod p (dX q))

/-- Derivative of a layer with respect to Z. -/
def dZ : Poly  Poly
  | zero => zero
  | one => zero
  | x => zero
  | z => one
  | sum p q => sum (dZ p) (dZ q)
  | prod p q => sum (prod (dZ p) q) (prod p (dZ q))

variable {X Z : Type}

/-- Fill the X-hole of a layer context. -/
def plugX : (p : Poly)  Elt X Z (dX p)  X  Elt X Z p
  | zero, e, _ => nomatch e
  | one, e, _ => nomatch e
  | x, .one, a => .x a
  | z, e, _ => nomatch e
```

```
  | sum p q, .inl e, a => .inl (plugX p e a)
  | sum p q, .inr e, a => .inr (plugX q e a)
  | prod p q, .inl (.pair e b), a => .pair (plugX p e a) b
  | prod p q, .inr (.pair b e), a => .pair b (plugX q e a)

/-- Fill the Z-hole of a layer context. -/
def plugZ : (p : Poly)  Elt X Z (dZ p)  Z  Elt X Z p
  | zero, e, _ => nomatch e
  | one, e, _ => nomatch e
  | x, e, _ => nomatch e
  | z, .one, c => .z c
  | sum p q, .inl e, c => .inl (plugZ p e c)
  | sum p q, .inr e, c => .inr (plugZ q e c)
  | prod p q, .inl (.pair e b), c => .pair (plugZ p e c) b
  | prod p q, .inr (.pair b e), c => .pair b (plugZ q e c)

/-- Every way to select an X in a layer, with its context. -/
def holesX : (p : Poly)  Elt X Z p  List (Elt X Z (dX p) × X)
  | zero, e => nomatch e
  | one, .one => []
  | x, .x a => [(.one, a)]
  | z, .z _ => []
  | sum p q, .inl e => (holesX p e).map fun (c, a) => (.inl c, a)
  | sum p q, .inr e => (holesX q e).map fun (c, a) => (.inr c, a)
  | prod p q, .pair a b =>
      (holesX p a).map (fun (c, v) => (.inl (.pair c b), v)) ++
      (holesX q b).map (fun (c, v) => (.inr (.pair a c), v))

/-- Every way to descend into a recursive child, with its layer context. -/
def holesZ : (p : Poly)  Elt X Z p  List (Elt X Z (dZ p) × Z)
  | zero, e => nomatch e
  | one, .one => []
  | x, .x _ => []
  | z, .z c => [(.one, c)]
  | sum p q, .inl e => (holesZ p e).map fun (s, c) => (.inl s, c)
  | sum p q, .inr e => (holesZ q e).map fun (s, c) => (.inr s, c)
  | prod p q, .pair a b =>
      (holesZ p a).map (fun (s, c) => (.inl (.pair s b), c)) ++
      (holesZ q b).map (fun (s, c) => (.inr (.pair a s), c))

/-- A context in a recursive type: a local context and a stack of steps,
the steps ordered from the hole outward. -/
structure Ctx (p0 : Poly) (X : Type) where
  local_ : Elt X (Mu p0 X) (dX p0)
  steps  : List (Elt X (Mu p0 X) (dZ p0))

/-- Rebuild the whole structure from a context and a replacement. -/
def plug {p0 : Poly} (c : Ctx p0 X) (a : X) : Mu p0 X :=
  c.steps.foldl (fun m s => .mk (plugZ p0 s m)) (.mk (plugX p0 c.local_ a))

/-- Every context of a structure, paired with the selected element. -/
partial def holes {p0 : Poly} (t : Mu p0 X) : List (Ctx p0 X × X) :=
```

```
  match t with
  | .mk e =>
    let here := (holesX p0 e).map fun (l, a) =>
      ({ local_ := l, steps := [] }, a)
    let below := (holesZ p0 e).flatMap fun (s, child) =>
      (holes child).map fun (c, a) => ({ c with steps := c.steps ++ [s] }, a)
    here ++ below

/-! Lists: F(X,Z) = 1 + X * Z -/
def listP : Poly := sum one (prod x z)

def ofList : List X  Mu listP X
  | [] => .mk (.inl .one)
  | a :: as => .mk (.inr (.pair (.x a) (.z (ofList as))))

partial def toList : Mu listP X  List X
  | .mk (.inl .one) => []
  | .mk (.inr (.pair (.x a) (.z t))) => a :: toList t

/-! Binary trees with leaves: F(X,Z) = X + Z * Z -/
def treeP : Poly := sum x (prod z z)

def leaf (a : X) : Mu treeP X := .mk (.inl (.x a))
def node (l r : Mu treeP X) : Mu treeP X := .mk (.inr (.pair (.z l) (.z r)))

partial def leaves : Mu treeP X  List X
  | .mk (.inl (.x a)) => [a]
  | .mk (.inr (.pair (.z l) (.z r))) => leaves l ++ leaves r

def sample : Mu listP Nat := ofList [1, 2, 3]

-- stack depth of each selection, and the selected element
#eval (holes sample).map fun (c, a) => (c.steps.length, a)
-- replace each selected element by 9
#eval (holes sample).map fun (c, _) => toList (plug c 9)
-- reconstruction with the original element restores the list
#eval (holes sample).all fun (c, a) => toList (plug c a) == [1, 2, 3]

partial def shape [ToString X] : Mu treeP X  String
  | .mk (.inl (.x a)) => toString a
  | .mk (.inr (.pair (.z l) (.z r))) => "(" ++ shape l ++ " " ++ shape r ++ ")"

def sampleTree : Mu treeP Nat := node (node (leaf 1) (leaf 2)) (leaf 3)
#eval (holes sampleTree).map fun (c, a) => (c.steps.length, a)
#eval (holes sampleTree).map fun (c, _) => leaves (plug c 9)
#eval (holes sampleTree).all fun (c, a) => leaves (plug c a) == [1, 2, 3]
-- constructor-preserving view: restoration of the whole tree
#eval shape sampleTree
#eval (holes sampleTree).map fun (c, _) => shape (plug c 9)
#eval (holes sampleTree).all fun (c, a) => shape (plug c a) == shape sampleTree


[(0, 1), (1, 2), (2, 3)]
```

```
[[9, 2, 3], [1, 9, 3], [1, 2, 9]]
true
[(2, 1), (2, 2), (1, 3)]
[[9, 2, 3], [1, 9, 3], [1, 2, 9]]
true
"((1 2) 3)"
["((9 2) 3)", "((1 9) 3)", "((1 2) 9)"]
true
```

References

[1] C. McBride. The Derivative of a Regular Type Is Its Type of One-Hole Contexts. Unpublished manuscript, 2001.

[2] P. Morris, T. Altenkirch, and C. McBride. Exploring the Regular Tree Types. In *Types for Proofs and Programs (TYPES 2004)*, LNCS 3839, pages 252–267. Springer, 2006.

[3] H. Goguen, C. McBride, and J. McKinna. Eliminating Dependent Pattern Matching. In *Algebra, Meaning, and Computation: Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, LNCS 4060, pages 521–540. Springer, 2006.

[4] G. Huet. Functional Pearl: The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[5] M. Abbott, T. Altenkirch, N. Ghani, and C. McBride. Derivatives of Containers. In *Typed Lambda Calculi and Applications (TLCA 2003)*, LNCS 2701, pages 16–30. Springer, 2003.

[6] M. Abbott, T. Altenkirch, N. Ghani, and C. McBride. ∂ for Data: Differentiating Data Structures. *Fundamenta Informaticae*, 65(1–2):1–28, 2005.

[7] M. Abbott, T. Altenkirch, and N. Ghani. Containers: Constructing Strictly Positive Types. *Theoretical Computer Science*, 342(1):3–27, 2005.

[8] T. Altenkirch and C. McBride. Generic Programming within Dependently Typed Programming. In *Generic Programming*, IFIP 115, pages 1–20. Kluwer, 2003.

[9] C. McBride and J. McKinna. Functional Pearl: I Am Not a Number, I Am a Free Variable. In *Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2004)*, pages 1–9. ACM, 2004.