The Derivative of a Regular Type is its Type of One-Hole Contexts

Extended Abstract

Conor McBride*

Abstract

Polymorphic regular types are tree-like datatypes generated by polynomial type expressions over a set of free variables and closed under least fixed point. The 'equality types' of Core ML can be expressed in this form. Given such a type expression T with x free, this paper shows a way to represent the one-hole contexts *for elements of x within elements of T, together with an operation which will plug an element of x into the hole of such a context. One-hole contexts are given as inhabitants of a regular type $\partial_x T$, computed* generically *from the syntactic structure of T by a mechanism better known as* partial differentiation*. The relevant notion of containment is shown to be appropriately characterized in terms of derivatives and plugging in. The technology is then exploited to give the one-hole contexts for sub-elements of recursive types in a manner similar to Huet's 'zippers'[Hue97].*

1 Introduction

Gérard Huet's delightful paper 'The Zipper' [Hue97] defines a representation of tree-like data decomposed into a subtree of interest and its surroundings. He shows us informally how to equip a datatype with an associated type of 'zippers'—one-hole contexts representing a tree with one subtree deleted. Zippers collect the subtrees forking off step by step from the path which starts at the hole and returns to the root. This type of contexts is thus independent of the particular tree being decomposed or the subtree in the hole. Decomposition is seen not as a kind of *subtraction*, an operation inherently troubled by the need to subtract only smaller things from larger, but as the *inversion* of a kind of *addition* (see figure 1).

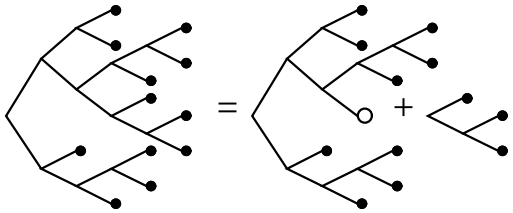


Figure 1: *a tree as a one-hole context 'plus' a subtree*

This paper exhibits a similar technique, defined more formally, which is generic for a large class of tree-like data structures—the 'regular datatypes'. These are essentially the 'equality types' of core ML, presented as polynomial type expressions closed under least fixed point. In particular, I will define and characterize two operations:

- on *types*, computing for each regular recursive datatype its type of one-hole contexts;
- on *terms*, computing a 'big' term by plugging a 'small' term into a one-hole context.

The first of these operations is given by recursion on the structure of the algebraic expressions which 'name' types. As I wrote down the rules corresponding to the empty type, the unit type, sums and products, I was astonished to find that Leibniz had beaten me to them by several centuries: they were exactly the rules of differentiation I had learned as a child.

*Department of Computer Science, University of Durham, c.t.mcbride@durham.ac.uk

1.1 Motivating Examples

How can we describe the one-hole contexts of a recursive type? Huet suggests that we regard them as journeys 'backwards' from hole to root, recording what we pass by on the way. I propose to follow his suggestion, except that I will start at the root and work my way 'forwards' to the hole. Both choices have their uses: 'backwards' is better for 'tree editing' applications, but the 'forwards' approach is conceptually simpler.

Consider such journeys within binary trees:

$$\mathbf{btree} \;=\; \mathsf{leaf} \mid \mathsf{node}\ \mathbf{btree}\ \mathbf{btree}$$

At each point, we are either at the hole or we must step further in—our journey is a list of steps, **list btree′** for some appropriate step type **btree′**, but what? At each step, we must record our binary choice of left or right, together with the **btree** we passed by. We may take

$$\mathbf{btree}' \;=\; \mathbf{bool}\times\mathbf{btree}$$

We can define the 'plugging in' operation $+$ as follows[1]

$$\frac{s \in \mathbf{btree}' \qquad t \in \mathbf{btree}}{s \lessdot t \in \mathbf{btree}}$$

$$\begin{array}{rcl} (\mathsf{true}, r) \lessdot t & \mapsto & \mathsf{node}\ t\ r \\ (\mathsf{false}, l) \lessdot t & \mapsto & \mathsf{node}\ l\ t \end{array}$$

$$\frac{ss \in \mathbf{list}\ \mathbf{btree}' \qquad t \in \mathbf{btree}}{ss + t \in \mathbf{btree}}$$

$$\begin{array}{rcl} [] + t & \mapsto & t \\ (s :: ss) + t & \mapsto & s \lessdot (ss + t) \end{array}$$

We might define **btree** more algebraically as the sum (i.e. 'choice') of the unique leaf with nodes pairing two subtrees—powers denote repeated product:

$$\mathbf{btree} = \mathbf{1} + \mathbf{btree}^{\mathbf{2}}$$

Correspondingly, we might write

$$\mathbf{btree}' = \mathbf{2}\times\mathbf{btree}$$

Now imagine similar journeys **list ttree′** for ternary trees:

$$\mathbf{ttree} = \mathbf{1} + \mathbf{ttree}^{\mathbf{3}}$$

Each step chooses one of *three* directions, and remembers *two* bypassed trees, so

$$\mathbf{ttree}' = \mathbf{3}\times\mathbf{ttree}^{\mathbf{2}}$$

It looks remarkably like the type of steps is calculated by *differentiating* the original datatype. In fact, this is exactly what is happening, as we shall see when we examine how to compute the description of the one-hole contexts for regular datatypes.

2 Presenting the Regular Types

In order to give a precise treatment of 'differentiating a datatype', we must be precise about the expressions which define them. In particular, the availability of fixed points requires us to consider the binding of fresh type variables. There is much activity in this area of research, but this is not the place to rehearse its many issues. The approach I take in this paper relativizes regular type expressions to finite sequences of available free names. I then explain how to interpret such expressions relative to an environment which interprets those names.

I choose to give inductive definitions in natural deduction style. Although this requires more space than the datatype declarations of conventional programming languages, they allow *dependent families* to be presented much more clearly: even 'nested' types [BM98, BP99] must be defined *uniformly* over their indices, whilst the full notion of family allows constructors to apply only at *particular* indices. I also give type signatures to defined functions in this way, preferring to present the universal quantification inherent in their type dependency via schematic use of variables, rather than by complex and inscrutable formulae.

[1] Huet would write $(s :: ss) + t \;\mapsto\; ss + (s \lessdot t)$

2.1 Sequences of Distinct Names

Let us presume the existence of an infinite[2] set **Name** of names, equipped with a decidable equality. We may think of **Name** as being 'string'. The set **NmSeq** may then be defined to contain the finite sequences of distinct names. Each such sequence can be viewed 'forgetfully' as a set.

$$\frac{}{\mathbf{Name} \in \mathbf{Set}} \qquad \frac{\Sigma \in \mathbf{NmSeq}}{\Sigma \in \mathbf{Set}}$$

$$\frac{}{\varepsilon \in \mathbf{NmSeq}} \qquad \frac{\Sigma \in \mathbf{NmSeq} \quad x \in \mathbf{Name} \quad x \notin \Sigma}{\Sigma; x \in \mathbf{NmSeq}}$$

I shall always use names in a well-founded manner. The 'explanation' of some name x from some sequence Σ will involve only 'prior' names—intuitively, those which have already been explained. It is therefore important to define for any such pair, the *restriction* $\Sigma \lfloor x$, being the sequence of names from Σ prior to x.

$$\frac{\Sigma \in \mathbf{NmSeq} \quad x \in \Sigma}{\Sigma \lfloor x \in \mathbf{NmSeq}}$$

$$\begin{array}{rcl} \Sigma; x \lfloor x & \mapsto & \Sigma \\ \Sigma; (y \backslash x) \lfloor x & \mapsto & \Sigma \lfloor x \end{array}$$

As equality on names is decidable, I shall freely allow names to occur nonlinearly in patterns. In order to recover the disjointness of patterns without recourse to prioritizing them, I introduce the notation $y \backslash x$ to mean 'any y except x'. Correspondingly, each clause of such a definition holds (schematically) as an equation. I nonetheless write $\mapsto$ to indicate a directed *computation* rule, reserving $=$ for equational *propositions*.

2.2 Describing the Regular Types

The regular types over given free names, or rather, the expressions which *describe* them are given by the inductive family **Reg** Σ in figure 2. Firstly, we embed[3] type variables Σ in **Reg** Σ, then we give the building blocks for polynomials and least fixed point.

$$\frac{\Sigma \in \mathbf{NmSeq}}{\mathbf{Reg}\ \Sigma \in \mathbf{Set}}$$

$$\frac{x \in \Sigma}{x \in \mathbf{Reg}\ \Sigma}$$

$$\frac{}{0 \in \mathbf{Reg}\ \Sigma} \qquad \frac{S, T \in \mathbf{Reg}\ \Sigma}{S + T \in \mathbf{Reg}\ \Sigma}$$

$$\frac{}{1 \in \mathbf{Reg}\ \Sigma} \qquad \frac{S, T \in \mathbf{Reg}\ \Sigma}{S \times T \in \mathbf{Reg}\ \Sigma}$$

$$\frac{F \in \mathbf{Reg}\ \Sigma; x}{\mu x.\, F \in \mathbf{Reg}\ \Sigma}$$

$$\frac{F \in \mathbf{Reg}\ \Sigma; x \quad S \in \mathbf{Reg}\ \Sigma}{F|x{=}S \in \mathbf{Reg}\ \Sigma} \qquad \frac{T \in \mathbf{Reg}\ \Sigma}{\underline{T}_x \in \mathbf{Reg}\ \Sigma; x}$$

Figure 2: *descriptions of regular types*

The last two constructors may seem mysterious. However, the redundancy they introduce will save us work when we come to *interpret* these descriptions of types. We could choose to interpret only **Reg** ε, the *closed* descriptions, keeping the open types underwater like the dangerous bits of an iceberg. This would require us to *substitute* for free names whenever they become exposed, for example, when expanding a fixed point. We would in turn be forced to take account of the equational properties of substitutions and prove closure properties with respect to them. I dispense with substitution.

The alternative I have adopted is to interpret *open* descriptions in *environments* which explain their free names. The two unusual constructors above perform the rôles of *definition* and *weakening*, respectively growing and shrinking the environment within their scope. Definition replaces substitution and weakening adds more free names without modifying 'old' descriptions. This avoids the propagation of substitutions through the syntactic structure and hence concerns about capture—we shall only ever interpret the 'value' of a variable in its binding-time environment.

[2] By 'infinite', I mean that I can always choose a fresh name if I need to create a new binding.

[3] In fact I am abusing notation by suppressing a constructor symbol.

2.3 Type Environments

A type environment for a given name sequence Σ associates each x in Σ with a type description over the prior names, $\Sigma \lfloor x$.

$$\frac{\Sigma \in \mathsf{NmSeq}}{\mathsf{Env}\ \Sigma \in \mathbf{Set}}$$

$$\frac{}{\varepsilon \in \mathsf{Env}\ \varepsilon}$$

$$\frac{\Sigma; x \in \mathsf{NmSeq} \quad \Gamma \in \mathsf{Env}\ \Sigma \quad S \in \mathsf{Reg}\ \Sigma}{\Gamma; x{=}S \in \mathsf{Env}\ \Sigma; x}$$

We may equip environments with operators for *restriction* (extracting the prefix prior to a given name) and *projection* (looking up a name's associated type description). $\Gamma \lfloor x$ is the environment which explains the free names in $\Gamma \cdot x$.

$$\frac{\Gamma \in \mathsf{Env}\ \Sigma \quad x \in \Sigma}{\Gamma \lfloor x \in \mathsf{Env}\ \Sigma \lfloor x} \qquad \frac{\Gamma \in \mathsf{Env}\ \Sigma \quad x \in \Sigma}{\Gamma \cdot x \in \mathsf{Reg}\ \Sigma \lfloor x}$$

$$\begin{array}{rcl} \Gamma; x{=}S \lfloor x & \mapsto & \Gamma \\ \Gamma; (y \backslash x){=}S \lfloor x & \mapsto & \Gamma \lfloor x \end{array}$$

$$\begin{array}{rcl} \Gamma; x{=}S \cdot x & \mapsto & S \\ \Gamma; (y \backslash x){=}S \cdot x & \mapsto & \Gamma \cdot x \end{array}$$

2.4 Interpreting the Descriptions

Now that we have the means to describe regular types, we must say which data are contained by the type with a given description, relative to an environment explaining its free names—we must give a semantics to the syntax. Figure 3 defines the interpretation $\Gamma \llbracket T \rrbracket$ inductively.[4]

Note that I have not supplied constructor symbols for the embedding rules corresponding to variable-lookup, definition and weakening. These apparent conflations of types, e.g. $\Gamma; x{=}S \llbracket x \rrbracket$ with $\Gamma \llbracket S \rrbracket$, are harmless, as the types tell us which embeddings are operative.

[4] The 'semantic brackets' do not represent a meta-level operation; they are intended to be a suggestive object-level syntax for a dependent family of types.

$$\frac{\Gamma \in \mathsf{Env}\ \Sigma \quad T \in \mathsf{Reg}\ \Sigma}{\Gamma \llbracket T \rrbracket \in \mathbf{Set}}$$

$$\frac{t \in \Gamma \lfloor x \llbracket \Gamma \cdot x \rrbracket}{t \in \Gamma \llbracket x \rrbracket}$$

$$\frac{s \in \Gamma \llbracket S \rrbracket}{\mathsf{inl}\ s \in \Gamma \llbracket S + T \rrbracket} \qquad \frac{t \in \Gamma \llbracket T \rrbracket}{\mathsf{inr}\ t \in \Gamma \llbracket S + T \rrbracket}$$

$$\frac{}{\langle\rangle \in \Gamma \llbracket 1 \rrbracket} \qquad \frac{s \in \Gamma \llbracket S \rrbracket \quad t \in \Gamma \llbracket T \rrbracket}{\langle s; t \rangle \in \Gamma \llbracket S \times T \rrbracket}$$

$$\frac{t \in \Gamma \llbracket F | x{=}\mu x.\, F \rrbracket}{\mathsf{con}\ t \in \Gamma \llbracket \mu x.\, F \rrbracket}$$

$$\frac{t \in \Gamma; x{=}S \llbracket F \rrbracket}{t \in \Gamma \llbracket F | x{=}S \rrbracket} \qquad \frac{t \in \Gamma \llbracket T \rrbracket}{t \in \Gamma; x{=}S \llbracket \underline{T}_x \rrbracket}$$

Figure 3: *the data in regular types*

2.5 Examples of Regular Types

Let us briefly examine some familiar types in this setting. We have the unit type $\mathsf{1}$, so we can use $+$ to build the booleans:

$$\begin{array}{rcll} \mathbf{bool} & \mapsto & \mathsf{1} + \mathsf{1} & (\in \mathsf{Reg}\ \Sigma,\ \text{any } \Sigma) \\ \mathrm{true} & \mapsto & \mathsf{inl}\ \langle\rangle & (\in \Gamma \llbracket \mathbf{bool} \rrbracket,\ \text{any } \Gamma) \\ \mathrm{false} & \mapsto & \mathsf{inr}\ \langle\rangle & (\in \Gamma \llbracket \mathbf{bool} \rrbracket,\ \text{any } \Gamma) \end{array}$$

We may use μ to build recursive datatypes like the natural numbers and our tree examples, again for any Σ and Γ, as a fresh bound variable (x below) can always be chosen:

$$\begin{array}{rcl} \mathbf{nat} & \mapsto & \mu x.\, \mathsf{1} + x \\ \mathrm{zero} & \mapsto & \mathsf{con}\ (\mathsf{inl}\ \langle\rangle) \\ \mathrm{suc}\, n & \mapsto & \mathsf{con}\ (\mathsf{inr}\ n) \end{array}$$

$$\begin{array}{rcl} \mathbf{btree} & \mapsto & \mu x.\, \mathsf{1} + x \times x \\ \mathrm{bleaf} & \mapsto & \mathsf{con}\ (\mathsf{inl}\ \langle\rangle) \\ \mathrm{bnode}\, l\, r & \mapsto & \mathsf{con}\ (\mathsf{inr}\ \langle l; r \rangle) \end{array}$$

$$\begin{array}{rcl} \mathbf{ttree} & \mapsto & \mu x.\, \mathsf{1} + x \times (x \times x) \\ \mathrm{tleaf} & \mapsto & \mathsf{con}\ (\mathsf{inl}\ \langle\rangle) \\ \mathrm{tnode}\, l\, m\, r & \mapsto & \mathsf{con}\ (\mathsf{inr}\ \langle l; \langle m; r \rangle\rangle) \end{array}$$

We may use *weakening* to define an operation which computes list types:

$$\begin{array}{rcl} \mathbf{list}\, T & \mapsto & \mu x.\, 1 + \underline{T}_x \times x \\ \mathrm{nil} & \mapsto & \mathsf{con}\ (\mathsf{inl}\ \langle\rangle) \\ x :: xs & \mapsto & \mathsf{con}\ (\mathsf{inr}\ \langle x; xs\rangle) \end{array}$$

The finitely branching trees may thus be given by

$$\begin{array}{rcl} \mathbf{ftree} & \mapsto & \mu x.\, \mathbf{list}\, x \\ \mathrm{fnode}\ ts & \mapsto & \mathsf{con}\ ts \end{array}$$

2.6 Subterm Orderings for Regular Types

Let us now define an inductive relation, $u \leq_x^T t$ for $u \in \Gamma\,[\![x]\!]$ and $t \in \Gamma\,[\![T]\!]$, characterizing the subterms of a term in T accounted for by x's in T. This relation characterizes T's rôle as a container of x's. I omit the obvious 'well-formedness' premises.

$$\frac{u \in \Gamma\,[\![x]\!]}{u \leq_x^x u}$$

$$\frac{u \leq_x^S s}{u \leq_x^{S+T} \mathsf{inl}\ s} \qquad \frac{u \leq_x^T t}{u \leq_x^{S+T} \mathsf{inr}\ t}$$

$$\frac{u \leq_x^S s}{u \leq_x^{S\times T} \langle s; t\rangle} \qquad \frac{u \leq_x^T t}{u \leq_x^{S\times T} \langle s; t\rangle}$$

$$\frac{u \leq_x^{F|y=\mu y.\,F} t}{u \leq_x^{\mu y.\,F} \mathsf{con}\ t}$$

$$\frac{u \leq_x^F t}{u \leq_x^{F|y=S} t} \qquad \frac{u \leq_x^S s \quad s \leq_y^F t}{u \leq_x^{F|y=S} t} \qquad \frac{u \leq_x^T t}{u \leq_x^{\underline{T}_y} t}$$

Observe that subterms accounted for by an x in a composite type are characterized by a rule for each component of that type. In particular, the rules for a definition $F|y{=}S$ find subterms due to x's in F and x's in S respectively, the latter occurring within subterms due to y's in F. Note that the latter rule exploits the fact that we may see s as inhabiting both $\Gamma\,[\![S]\!]$ and $\Gamma; y{=}S\,[\![y]\!]$.

The effect of local type variables is thus reflected at their place of *binding*, rather than at their places of *use*.[5]

The significance of the phrase 'accounted for by the x's in T' is shown by this simple example:

If we take	$\Gamma = \varepsilon; x{=}\mathrm{bool}; y{=}x$
we can derive	$\mathrm{true} \leq_x^{x\times y} \langle \mathrm{true}; \mathrm{false}\rangle$
but not	$\mathrm{false} \leq_x^{x\times y} \langle \mathrm{true}; \mathrm{false}\rangle$

The point is that false has type x on account of the x in this particular Γ's definition of y, whereas, irrespective of Γ, we can always derive

$$s \leq_x^{x\times y} \langle s; t\rangle$$

However, discharging the definition of y in our example:

Taking	$\Gamma = \varepsilon; x{=}\mathrm{bool}$
we can derive	$\mathrm{true} \leq_x^{x\times y|y=x} \langle \mathrm{true}; \mathrm{false}\rangle$
and	$\mathrm{false} \leq_x^{x\times y|y=x} \langle \mathrm{true}; \mathrm{false}\rangle$

Both x's are accounted for by the indicated type, and the derivations follow respectively by the two definition rules.

A similar phenomenon occurs if we try to specialize this ordering to the x's contained by a $\mathbf{list}\ x$, that is, to determine when

$$t \leq_x^{\mu y.\,1+\underline{x}_y\times y} ts$$

Only the rule for fixed points applies, showing this derived rule to be complete:

$$\frac{t \leq_x^{(1+\underline{x}_y\times y)|y=\mu y.\,1+\underline{x}_y\times y} tts}{t \leq_x^{\mathbf{list}\ x} \mathsf{con}\ tts}$$

The premise can only be further simplified by the definition rules: the direct rule finds only the head of the list in x; the indirect rule's second premise finds only the tail in y, and the first demands that we search for an x within that tail. We thus derive (and show complete) the two rules we might have hoped for:

[5] It would be interesting to investigate a notion of subterm with a single rule for definitions capturing 'x-subterms within F-term t, remembering that y really means S': however, we would pay with extra complexity in the rule for variables, and with the need to carry extra environmental information explicitly.

$$\frac{}{t \leq_x^{\mathbf{list}\ x}\ (t :: ts)} \qquad \frac{t \leq_x^{\mathbf{list}\ x}\ ts}{t \leq_x^{\mathbf{list}\ x}\ (s :: ts)}$$

We may exploit this 'containment' relation to define the usual subterm relation for recursive types, namely $\leq_{\mu x.\, F}$ for $\Gamma \llbracket \mu x.\, F \rrbracket$, as follows:

$$\frac{}{u \leq_{\mu x.\, F} u} \qquad \frac{u \leq_{\mu x.\, F} s \qquad s \leq_x^F t}{u \leq_{\mu x.\, F} \mathsf{con}\, t}$$

That is, to find a subterm of $\mathsf{con}\, t \in \Gamma \llbracket \mu y.\, F \rrbracket$, either stop where you are, or move down one level to an x contained by F and repeat. $\leq_x^F$ cannot go further than one level, because x is *free* in F!

With a little work, we can specialize these rules for $\mathbf{btree}$ to

$$\frac{}{t \leq_{\mathbf{btree}} t}$$

$$\frac{t \leq_{\mathbf{btree}} l}{t \leq_{\mathbf{btree}} (\mathrm{bnode}\, l\, r)} \qquad \frac{t \leq_{\mathbf{btree}} r}{t \leq_{\mathbf{btree}} (\mathrm{bnode}\, l\, r)}$$

3 One-Hole Contexts

Let us now turn to the generic representation of one-hole contexts for recursive regular types. We shall first need to see what 'one step' is.

The immediate recursive subterms in some $\mathsf{con}\ t$ of type $\Gamma \llbracket \mu x.\, F \rrbracket$ are those subterms of t which correspond to the occurrences of x in F—recall that t has type $\Gamma; x{=}\mu x.\, F \llbracket F \rrbracket$. We therefore need to describe the one-hole contexts for x's in F's. Since F may itself contain fixed points, the primitive operation we need to define will compute for F the type of one-hole contexts for any of its free names. This operation is exactly *partial differentiation*.

3.1 Partial Differentiation

The partial derivative of a regular type description T with respect to a free name x is computed by structural recursion over the syntax of T.[6] It is defined in figure 4.

$$\frac{x \in \Sigma \qquad T \in \mathbf{Reg}\ \Sigma}{\partial_x T \in \mathbf{Reg}\ \Sigma}$$

$$\begin{array}{rcl}
\partial_x x & \mapsto & 1 \\
\partial_x (y \backslash x) & \mapsto & 0 \\
\partial_x 0 & \mapsto & 0 \\
\partial_x (S + T) & \mapsto & \partial_x S \ +\ \partial_x T \\
\partial_x 1 & \mapsto & 0 \\
\partial_x (S \times T) & \mapsto & \partial_x S \times T \ +\ S \times \partial_x T \\
\partial_x (\mu y.\, F) & \mapsto & \mu z.\, \underline{\partial_x F\,|y{=}\mu y.\, F}_z + \\
 & & \quad \underline{\partial_y F\,|y{=}\mu y.\, F}_z \times z \\
\partial_x (F|y{=}S) & \mapsto & \partial_x F\,|y{=}S \ +\ \partial_y F\,|y{=}S \times \partial_x S \\
\partial_x \underline{T}_x & \mapsto & 0 \\
\partial_x \underline{T}_{y \backslash x} & \mapsto & \underline{\partial_x T}_y
\end{array}$$

Figure 4: *partial differentiation*

The first six lines are familiar from calculus. For one-hole contexts, they tell us that

- an x contains one x in trivial surroundings
- a y other than x contains no x
- constants contain no x
- we find an x in an $S + T$ in either an S or a T
- we find an x in an $S \times T$ either (left) in the S, passing the T, or (right) in the T passing the S

Note that partial differentiation is independent of environments: it is defined on the *syntax* of types alone. Just like the subterm ordering, it takes no account of the possibility that some y might, for some Γ, expand in terms of x. However, partial differentiation is the basic tool by which total differentiation is constructed, and this is exactly what we need when we go under a binder and become liable to encounter *local* names which potentially do conceal x's.

The rule for definitions again handles local variables at their place of *binding*, summing the types of contexts for

[6] It is depressing how few mathematics teachers deign to impart this vital clue to calculus students, giving them rules but no *method.* I learned differentiation from the pattern matching algorithm given in [McB70].

x's occurring directly in F, and indirectly in an S buried within a y. In conventional calculus, this is effectively the 'chain rule' extended to functions of *two* arguments

$$\frac{\partial}{\partial x} f(a,b) = (\frac{\partial}{\partial u} f(u,v) \frac{\partial a}{\partial x} + \frac{\partial}{\partial v} f(u,v) \frac{\partial b}{\partial x})|_{(u,v)=(a,b)}$$

in the special case where a is x.

Once we know how to differentiate definitions, we can make the leap to fixed points. If we expand the fixed point and apply the 'chain rule', we obtain:

$$\begin{array}{rcl} \partial_x(\mu y.\, F) & = & \partial_x(F|y{=}\mu y.\, F) \\ & = & \partial_x F\,|y{=}\mu y.\, F + \\ & & \partial_y F\,|y{=}\mu y.\, F \times \partial_x(\mu y.\, F) \end{array}$$

It is tempting to solve this recursive equation with a least fixed point, but does this give the correct type of one-hole contexts? Yes: every x inside a $\mu y.\, F$ must be a piece of 'payload data' attached to some y-node buried at a *finite* depth. Hence our journey takes us either to an x at the outermost node and stops—hence the $\partial_x F$ in the 'base case'—or to a subnode and onwards—hence the $\partial_y F$ in the 'step case': it must stop eventually. We must weaken at z, for z is not free for F. Our journey is clearly *linear*: the body of the fixed point is *syntactically* linear in z. Indeed, the following set isomorphism holds, showing that our journey is a list of 'steps' with a 'tip':

$$\Gamma\,[\![\mu z.\, \underline{T}_z \;+\; \underline{S}_z \times z]\!] \cong \Gamma\,[\![(\mathbf{list}\; S) \times T]\!]$$

The rules for differentiating an explicit weakening simply short-circuit the process if the name we seek is that being excluded.

3.2 Examples of Derivatives

Before we develop any more technology, let us check that partial differentiation is giving us the kind of answers we expect. Of course, the types we get back will contain lots of '$0+$' and '$1\times$', but the usual algebraic laws which simplify such expressions hold as set isomorphisms. It is also not hard to show that a recursive type with no base cases is empty:

$$\textit{If}\ \Gamma\,[\![F|x{=}0]\!] \cong 0\ \textit{then}\ \Gamma\,[\![\mu x.\, F]\!] \cong 0$$

Writing T^n for the n-fold product of T's and n for the sum of n 1's, we can check (by induction on n), that for $n \geq 0$

$$\Gamma\,[\![\partial_x x^n]\!] \cong n \times x^{n-1}$$

Viewed as a one-hole context for x^n, the n tells us which x the hole is at, while the x^{n-1} records the remaining x's.

A more involved example finds an x within a list of x's

$$\begin{array}{rcl} \partial_x \mathbf{list}\; x & \cong & \partial_x(\mu y.\, 1 \;+\; x \times y) \\ & \cong & \mu z.\, \underline{(\partial_x(1 \;+\; x \times y))|y{=}\mathbf{list}\; x}_z \;+ \\ & & \quad \underline{(\partial_y(1 \;+\; x \times y))|y{=}\mathbf{list}\; x}_z \times z \\ & \cong & \mu z.\, \underline{y|y{=}\mathbf{list}\; x}_z \;+\; \underline{x|y{=}\mathbf{list}\; x \times z}_z \\ & \cong & (\mathbf{list}\; x) \times (\mathbf{list}\; x) \end{array}$$

We would indeed hope that a one-hole context for a list element is a pair of lists—the prefix and the suffix!

Amusingly, the following power series resembles $\mathbf{list}\; x$,

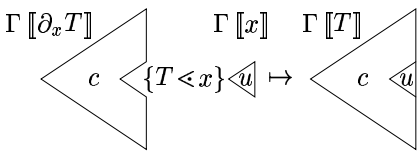
$$1 + x + x^2 + x^3 + \cdots = \frac{1}{1-x} \qquad (\textit{for}\ |x| < 1)$$

and conventional calculus tells us that

$$\frac{\partial}{\partial x}\left(\frac{1}{1-x}\right) = \left(\frac{1}{1-x}\right)^2$$

3.3 Plugging In

Given a one-hole context in $\partial_x T$ and an x, we should be able to construct a T by plugging the x in the hole. That is, we need an operation which behaves thus:



$$\frac{c \in \Gamma\,[\![\partial_x T]\!] \qquad u \in \Gamma\,[\![x]\!]}{c\ \{T \lessdot x\}\ u \in \Gamma\,[\![T]\!]}$$

$$\begin{array}{rcl}
\langle\rangle\ \{x \lessdot x\}\ u & \mapsto & u \\
\mathsf{inl}\ c\ \{S+T \lessdot x\}\ u & \mapsto & \mathsf{inl}\ (c\ \{S \lessdot x\}\ u) \\
\mathsf{inr}\ c\ \{S+T \lessdot x\}\ u & \mapsto & \mathsf{inr}\ (c\ \{T \lessdot x\}\ u) \\
\mathsf{inl}\ \langle c; t\rangle\ \{S \times T \lessdot x\}\ u & \mapsto & \langle c\ \{S \lessdot x\}\ u; t\rangle \\
\mathsf{inr}\ \langle s; c\rangle\ \{S \times T \lessdot x\}\ u & \mapsto & \langle s; c\ \{T \lessdot x\}\ u\rangle \\
\mathsf{con}\ (\mathsf{inl}\ c)\ \{\mu y.\, F \lessdot x\}\ u & \mapsto & \mathsf{con}\ (c\ \{F \lessdot x\}\ u) \\
\mathsf{con}\ (\mathsf{inr}\ \langle c; cs\rangle)\ \{\mu y.\, F \lessdot x\}\ u & \mapsto & \mathsf{con}\ (c\ \{F \lessdot y\}\ (cs\ \{\mu y.\, F \lessdot x\}\ u)) \\
\mathsf{inl}\ c\ \{F|y{=}S \lessdot x\}\ u & \mapsto & c\ \{F \lessdot x\}\ u \\
\mathsf{inr}\ \langle c_y; c_x\rangle\ \{F|y{=}S \lessdot x\}\ u & \mapsto & c_y\ \{F \lessdot y\}\ (c_x\ \{S \lessdot x\}\ u) \\
c\ \left\{\underline{T}_{y \setminus x} \lessdot x\right\}\ u & \mapsto & c\ \{T \lessdot x\}\ u
\end{array}$$

Figure 5: *plugging in*

How are we to define this operation? T should tell us the shape of the term we are trying to build; c should tell us which path to take and also supply the subterms corresponding to the 'off-path' components of pairs. In effect, the operation proceeds by structural recursion on c, but its flow of control involves a primary case analysis on T. Of course, we need only consider cases where $\partial_x T$ is not $\mathsf{0}$. The definition is in figure 5.

3.4 Subtrees in Recursive Regular Types

The ∂_x operation picks out x's from *containers* for x's. As suggested above, this gives us the tool we need to pick out *subtrees* from our tree-like *recursive* datatypes, interpreting t as a container for the immediate subtrees of $\mathsf{con}\ t$. Hence our intuition that contexts for recursive type T inhabit $\mathbf{list}\ T'$ can be made rigorous. The 'step type' for trees in $\mu x.\, F$ is

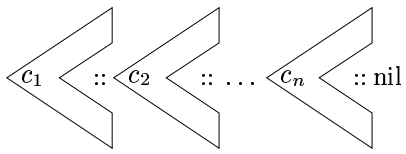
$$\partial_x F\,|x{=}\mu x.\, F$$

The one-hole contexts for $\mu x.\, F$ thus inhabit $\mathbf{sub}\ \mu x.\, F$ given by[7]

$$\frac{\mu x.\, F \in \mathbf{Reg}\ \Sigma}{\mathbf{sub}\ \mu x.\, F \in \mathbf{Reg}\ \Sigma}$$

$$\mathbf{sub}\ \mu x.\, F \quad \mapsto \quad \mathbf{list}\ (\partial_x F\,|x{=}\mu x.\, F)$$

Informally, an inhabitant of $\mathbf{sub}\ \mu x.\, F$ looks like



A few brief calculations reassure us that

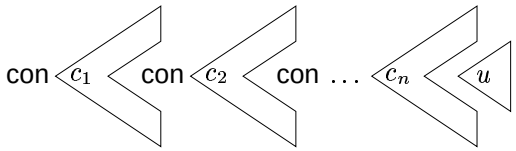
$$\begin{array}{rcl}
\Gamma\,[\![\mathbf{sub}\ (\mathbf{list}\ x)]\!] & \cong & \Gamma\,[\![\mathbf{list}\ x]\!] \\
\textit{hence}\ \Gamma\,[\![\mathbf{sub}\ (\mathbf{sub}\ \mu x.\, F)]\!] & \cong & \Gamma\,[\![\mathbf{sub}\ \mu x.\, F]\!] \\
\Gamma\,[\![\mathbf{sub}\ \mathbf{btree}]\!] & \cong & \Gamma\,[\![\mathbf{list}\ (2 \times \mathbf{btree})]\!] \\
\Gamma\,[\![\mathbf{sub}\ \mathbf{ttree}]\!] & \cong & \Gamma\,[\![\mathbf{list}\ (3 \times \mathbf{ttree}^2)]\!] \\
\Gamma\,[\![\mathbf{sub}\ \mathbf{ftree}]\!] & \cong & \Gamma\,[\![\mathbf{list}\ (\mathbf{list}\ \mathbf{ftree})^2]\!]
\end{array}$$

The corresponding notion of 'addition' has the signature

$$\frac{cs \in \Gamma\,[\![\mathbf{sub}\ \mu x.\, F]\!] \qquad u \in \Gamma\,[\![\mu x.\, F]\!]}{cs +_{\mu x.\, F} u \in \Gamma\,[\![\mu x.\, F]\!]}$$

[7] Abuse of notation—$\mathbf{sub}$ is really an operation on F

For the 'cs' depicted above and an appropriate 'u', we expect $cs +_{\mu x.\, F} u$ to be



Of course, $+_{\mu x.\, F}$ is defined by iterating $\{F \lessdot x\}$ over the list[8]:

$$\begin{array}{rcl} \mathsf{nil} +_{\mu x.\, F} u & \mapsto & u \\ (c :: cs) +_{\mu x.\, F} u & \mapsto & \mathsf{con}\,(c\ \{F \lessdot x\}\ (cs +_{\mu x.\, F} u)) \end{array}$$

Observe, for example, that $+_{\mathbf{nat}}$ really behaves like 'plus', whilst $+_{\mathbf{list}\ x}$ is effectively 'append'. It is not hard to show in general that $\mathbf{sub}\,\mu x.\, F$ is a *monoid* under 'append' and that $+_{\mu x.\, F}$ is an action of that monoid on $\mu x.\, F$.

3.5 Subterms and Derivatives

The relationship between the containment orderings and derivatives is an intimate one, and so too is that between the subtree orderings and the types computed by $\mathbf{sub}$. In effect, the containment ordering is exactly that *induced* by plugging in, while the subtree ordering on $\mu x.\, F$ is *induced* by $+_{\mu x.\, F}$. What we have done is to give a concrete representation for the witnesses to these relational properties. We may prove the following theorems:

Theorem (containment)
For $u \in \Gamma\,[\![x]\!]$ *and* $t \in \Gamma\,[\![T]\!]$*:*
$u \leq_x^T t \ \Leftrightarrow\ \exists c \in \Gamma\,[\![\partial_x T]\!]\,.\, c\ \{T \lessdot x\}\ u = t$

Theorem (subtree)
For $u, t \in \Gamma\,[\![\mu x.\, F]\!]$*:*
$u \leq_{\mu x.\, F} \Leftrightarrow\ \exists cs \in \Gamma\,[\![\mathbf{sub}\,\mu x.\, F]\!]\,.\, cs +_{\mu x.\, F} u = t$

The proofs of these theorems are easy inductions, on derivations for the $\Rightarrow$ direction, and on the structure of c or cs for the $\Leftarrow$ direction. It is moreover the case that distinct c's or cs's on the right give rise to distinct derivations on the left, and vice versa. I omit the details for reasons of space.

[8] Again, Huet would write
$(c :: cs) +_{\mu x.\, F} u \ \mapsto\ cs +_{\mu x.\, F} (\mathsf{con}\,(c\ \{F \lessdot x\}\ u))$

4 Towards an Implementation

Recent extensions to the HASKELL type system, such as Jansson and Jeuring's POLYP have begun to realise the potential of *generic programming* for the development of highly reusable code, instantiable for a wide class of datatypes and characterised by equally generic theorems [JJ97, BJJM98]. However, these systems show no sign of allowing operations like ∂_x which compute *types* generically by recursion over a closed syntax of type expressions, crucially regarding type variables as *concrete* objects.

In a *dependent* type theory supporting inductive *families* of datatypes [Dyb91], syntaxes such as $\mathbf{Reg}\,\Sigma$ can be represented as ordinary *data* which can then be used to index the families like $\Gamma\,[\![T]\!]$ which *reflect* them. ∂_x becomes merely an operation on data, requiring no extension to the computational power of the theory. A programming language based on such a type theory, as envisaged in [McB99], would seem to be a promising setting in which to implement the technology described in this paper. This work is well under way in the Computer-Assisted Reasoning Group at Durham.

The payoffs from such an implementation seem likely to be substantial, extending far beyond applications to editing trees. A library of generic tools for working with contexts would allow us to define functions in terms of these higher-level structures, manipulating data in large chunks, rather than one constructor at a time. Furthermore, in a dependently typed setting, a richer structure on data is *ipso facto* a richer structure on the indices of datatypes.

It also seems highly desirable to extend the class of datatypes for which one-hole contexts can be manipulated generically beyond the regular types, perhaps to include indexed families themselves. A concrete representation of one-hole contexts for a syntax with binding has much to offer both metaprogramming and metatheory.

5 Conclusions and Future Work

This paper has shown that the *one-hole contexts* for elements contained in a polymorphic regular type can be represented as the inhabitants of the regular type computed from the original by *partial differentiation*. This technology has been used to characterize data structures equivalent to Huet's 'zippers'—one-hole contexts for the subtrees of trees inhabiting arbitrary recursive types in that class. The operations which plug appropriate data into the holes of such contexts have also been exhibited.

While this connection seems unlikely to be a mere coincidence, it is perhaps surprising to find a use for the laws of the infinitesimal calculus in such a discrete setting. There is no obvious notion of 'tangent' or of 'limit' for datatypes which might connect with our intuitions from school mathematics. Neither can I offer at present any sense in which 'integration' might mean more than just 'differentiation backwards'.

One observation does, however, seem relevant: the syntactic operation of differentiating an expression with respect to x generates an approximation to the change in value of that expression by summing the contributions generated by varying *each* of the x's in the expression *in turn*. The derivative is thus the sum of terms corresponding to each one-hole context for an x in the expression. Perhaps the key to the connection can be found by focusing not on what is being infinitesimally varied, but on what, for the sake of a *linear* approximation to the curve, is being kept the same.

Apart from the implementation of this technology, and the development of a library of related generic utilities, this work opens up a host of fascinating theoretical possibilities—one only has to open one's old school textbooks almost at random and ask 'what does this mean for datatypes?'.

There must surely also be a relationship between this work and Joyal's general characterization of 'species of structure' in terms of their Taylor series[Joy87]. For regular types, $\partial_x^2 T$ makes a hole for a second x in a one-hole context from $\partial_x T$. More generally, $\partial_x^n T$ gives all the n-hole contexts for x's in T's in each of the $n!$ orders with which they can be found. This strong resonance with Taylor series seems worthy of pursuit and is an active topic of research.

In summary, the establishment of the connection between contexts and calculus is but the first step on a long road—who knows where it will end?

Acknowledgements

I feel very lucky to have stumbled on this interpretation of differentiation for datatypes with such potential for both utility and fascination, and has been intriguing me since the day I made the connection (whilst changing trains at Shrewsbury). This work has benefited greatly from hours spent on trains, and enthusiastic discussions with many people, most notably Konstantinos Tourlas, James McKinna, Alex Simpson, Daniele Turi, Martin Hofmann, Thorsten Altenkirch and, especially, Peter Hancock.

References

[BJJM98] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming—an introduction. In S. Doaitse Sweierstra, Pedro R. Henriques, and José N. Oliveira, editors, *Advanced Functional Programming, Third International Summer School (AFP '98); Braga, Portugal*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag, 1998.

[BM98] Richard Bird and Lambert Meertens. Nested datatypes. In *Mathematics of Program Construction*, volume 1422 of *LNCS*, pages 52–67. Springer-Verlag, 1998.

[BP99] Richard Bird and Ross Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–92, 1999.

[Dyb91] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*. CUP, 1991.

[Hue97] Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[JJ97] Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Proceedings of POPL '97*, pages 470–482. ACM, January 1997.

[Joy87] André Joyal. Foncteurs analytiques et espèces de structures. In *Combinatoire énumerative*, volume 1234 of *LNM*, pages 126–159. Springer-Verlag, 1987.

[McB70] Fred McBride. *Computer Aided Manipulation of Symbols*. PhD thesis, Queen's University of Belfast, 1970.

[McB99] Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.