Semantic Domain

Monday, February 15, 2021

Five (and a Half) Derivatives in Language Theory

Thanks to the influence of machine learning, differentiating programs has become a big business. However, there are a lot of other things in programming language theory which are also derivatives, and people sometimes get confused about the inter-relationships. So I decided to lay out some of them and some of their interconnections.

1. Brzozowski Derivatives

The first kind of derivative is the humble, yet beloved, Brzozowski derivative. Given a character set Σ and a regular language $L \subseteq \Sigma^*$, the Brzozowski derivative of L with respect to the character $c \in \Sigma$ is

$$\delta_c(L) = \{w \in \Sigma^* \ | \ | \ c \cdot w \in L\}$$

That is, it's the subset of L prefixed with c, minus the leading character. It can be defined as a transformations on regular expressions r as follows:

$$\begin{array}{lcll}
\delta_c(0) & = & \bot & \\
\delta_c(r_1 + r_2) & = & \delta_c(r_1) + \delta_c(r_2) & \\
\delta_c(c') & = & \epsilon & \text{when } c = c' \\
\delta_c(c') & = & \bot & \text{when } c \neq c' \\
\delta_c(\epsilon) & = & \bot & \\
\delta_c(r_1 \cdot r_2) & = & \delta_c(r_1) \cdot r_2 & \text{when } r_1 \text{ not nullable} \\
\delta_c(r_1 \cdot r_2) & = & \delta_c(r_1) \cdot r_2 + \delta_c(r_2) & \text{when } r_1 \text{ nullable} \\
\delta_c(r*) & = & \delta_c(r) \cdot r* &
\end{array}$$

The Brzozowski derivative is called a derivative for two reasons. First, it is linear with respect to $+$ on regular expressions -- i.e., $\delta_c(r_1 + r_2) = \delta_c(r_1) + \delta_c(r_2)$. Second, it satisfies the derivative rule for monomials $\delta_c(c^{n+1}) = c^n$. You may recall that the high-school rule for derivatives says $\delta_c(c^{n+1}) = n \times c^n$, but recall that in regular expressions, $r + r = r$, so $n \times r = r$.

However, this is a weak notion of derivative, which satisfies few of properties with other derivatives. The reason for this is that the rule for products does not match the product rule for high-school derivatives -- if the Brzozowski derivative satisfied the high school rule, it would be

$$\delta_c(r_1 \cdot r_2) = \delta_c(r_1) \cdot r_2 + r_1 \cdot \delta_c(r_2)$$

2. Differential algebras

If we try to change the definition of Brzozowski derivative to match the high school rule, we get what is called a *differential algebra*. A differential algebra is a semiring S, plus a collection of unary functions $\delta_i : S \to S$ which are called the *derivations*. The axioms for derivations mirror the rules for differentiating polynomials -- the motivating model for differential algebras is a semiring of polynomials with addition and multiplication of polynomials as the semiring structure, and the derivation is the derivatives with respect to the polynomial's variable. The rules we get are:

$$\begin{array}{lcll}
\delta_c(0) & = & 0 & \\
\delta_c(r_1 + r_2) & = & \delta_c(r_1) + \delta_c(r_2) & \\
\delta_c(c') & = & \epsilon & \text{when } c = c' \\
\delta_c(c') & = & \bot & \text{when } c \neq c' \\
\delta_c(\epsilon) & = & 0 & \\
\delta_c(r_1 \cdot r_2) & = & \delta_c(r_1) \cdot r_2 + r_1 \cdot \delta_c(r_2) & \\
\delta_c(r*) & = & r * \cdot \delta_c(r) \cdot r* &
\end{array}$$

Note that the only things that changed is the clause for concatenation and the Kleene star. The axioms of a differential algebra actually say nothing about Kleene star, because they are specified for

About Me

Neel Krishnaswami

View my complete profile

semirings, but to handle full regular languages you need to change the definition so that the differential respects the equation $r* = \epsilon + r \cdot r*$. If memory serves, Dexter Kozen has studied this extension. (Even if memory doesn't serve, he's a safe bet...!)

If you look at this definition for a while, you realize that the original Brzozowski derivative removes a single character c from the *front* of a string, and the differential removes a single c from *anywhere* in the string.

For regular algebras, this is a peculiar thing to want to do, but it suddenly becomes a lot more interesting when we move from semirings of polynomials to semiring categories of polynomial functors.

3. Derivatives of Data Types

One interesting factoid is that sets "look like" a semiring. The Cartesian product $A \times B$ and the disjoint union $A + B$ obviously don't satisfy the semiring equations, but if you replace the equality symbol with isomorphisms, then they do. If you take this analogy seriously and try categorify the notion of a semiring, we get what is variously called a *rig category*, *semiring category*, or *bimonoidal category*.

Instead of a set plus two binary operations, we have a category with two monoidal structures with the semiring equations becoming isomorphisms. The exact commuting diagrams needed were worked out in the early 70s, independently by Laplaza and Kelly.

One semiring category very useful category for programming is the category of *polynomial functors*. Most inductive datatypes like lists, binary trees, and so on can be understood as the least fixed point of sum and products. We can formalise this idea by giving a grammar of sums, products and

$$\begin{array}{rcl} F, G & ::= & \mathrm{Id} \\ & | & \underline{0} \\ & | & F \oplus G \\ & | & \underline{1} \\ & | & F \otimes G \\ & | & K(A) \quad \text{where } A \in \mathrm{Set} \end{array}$$

Every term of this grammar can be interpreted as a functor:

$$\begin{array}{lcl} [\![F]\!] & : & Set \to Set \\ [\![\mathrm{Id}]\!] & = & X \mapsto X \\ [\![\underline{0}]\!] & = & X \mapsto [\![F \oplus G]\!] \quad = \quad X \mapsto [\![F]\!]\, X + [\![G]\!]\, X \\ [\![F \otimes G]\!] & = & X \mapsto [\![F]\!]\, X \times [\![G]\!]\, X \\ [\![K(A)]\!] & = & X \mapsto A \end{array}$$

So you can see that $\oplus$ is interpreted as the coproduct functor, $\otimes$ is the product functor, $K(A)$ is the constant functor, and Id is interpreted as the identity functor.

The least fixed point of one of these polynomials is an inductive type. So to model lists of type X, we would define the list functor

$$\mathrm{ListF} \triangleq K(1) \oplus (K(X) \otimes \mathrm{Id})$$

and then take its least fixed point.

Since we have a semiring category, we can ask if it has a categorical analogue of a derivation, to give it differential structure. And these polynomials do! The derivative with respect to the variable Id can be defined as

$$\begin{array}{lcl} \delta(K(A)) & = & 0 \\ \delta(\underline{0}) & = & 0 \\ \delta(F \oplus G) & = & \delta(F) \oplus \delta(G) \\ \delta(\mathrm{Id}) & = & 1 \\ \delta(\underline{1}) & = & 0 \\ \delta(F \otimes G) & = & \delta(F) \otimes G \oplus F \otimes \delta(G) \end{array}$$

If you have a functor for a binary tree:

$$\mathrm{TreeF} \triangleq \underline{1} \oplus (\mathrm{Id} \otimes \mathrm{Id})$$

Then the differential is:

$$\delta(\mathrm{TreeF}) = (\underline{1} \otimes \mathrm{Id}) \oplus (\mathrm{Id} \otimes \underline{1})$$

This tells you whether the left or the right position was chosen. The utility of this is that given an element $x \in X$, and you have an element of $\delta(F)\ X$, then you can "fill in the gap" to get an element of X. That is, you can define:

$$\mathit{fill}_F : \delta(F)\ X \times X \to F\ X$$

If you take X to be the type of trees $\mu(\mathrm{TreeF})$, this gives you an indication of whether to take the left or right subtree, plus the tree you didn't follow. Then a list of these things gives you a path into a tree -- which is just Huet's famous *zipper*.

The exposition so far largely glosses Conor McBride's famous paper *The Derivative of a Regular Type is the Type of its One-Hole Contexts*.

The precise connection to linearity and derivatives was explored by Abbot, Altenkirch, Ghani and McBride in their paper *Derivatives of Containers*, using significantly more categorical machinery. In their paper, Altenkirch *et al* explain how the fill operation is really a *linear* function

$$\mathit{fill}_F : \delta(F)\ X \otimes X \multimap F\ X$$

and how $\delta(F)$ can be viewed as giving a kind of tangent for F at X.

4. Differential Linear Logic

The appearance of linear types here should be delightful rather than shocking. We think of the ususal derivative as associating a tangent vector space to each point of an original space, and some of the primordial models of linear type theory are finite-dimensional vector spaces and linear transformations (a model of the multiplicative-additive fragment of classical linear logic, indeed also of the exponentials when the field the vector space is over is finite), and Banach spaces and short maps, a simple model of intuitionistic linear logic including the full exponential.

As a result, we should expect that we should be able to find a way of making differentiation makes sense within linear type theory, and since a type theory is the initial model of a class of models, this means that derivative-like constructions are something we should look for any time linearity arises.

The pure form of this recipe can be found in Erhard and Regnier's differential linear logic. One of the really cool features of this line of work is that derivatives arise as an extension to the usual structural rules for the derivatives -- you just adjust how to treat variables and derivatives appear like magic. One fact about this line of work is that it has tended to focus on proof nets rather than sequent calculus, which made it less accessible than one would like. What solved this issue for me was Marie Kerjean's *A Logical Account for Linear Partial Differential Equations*.

I do want to admit that despite starting off this section saying you shouldn't be shocked, actually I am -- this stuff really seems like sorcery to me. So even though I believe intellectually that everything should work out like this, I'm still astounded that it does. (It's rather like the feeling I get whenever I run a program I did a machine-checked correctness proof of.)

5. Automatic Differentiation

These days, the most famous kind of differentiation in language theory is, of course, *automatic differentiation*, which takes a program and rewrites it to produce a new program computing the derivative of the original program. Automatic differentiation comes in two main varieties, called *forward mode* and *reverse mode*.

The idea of forward mode is incredibly simple -- you just treat real numbers as an abstract datatype, and replace your original real number implementation with a new one based on *dual numbers*. Intuitively, a dual number is a pair $a + b\epsilon$, where ϵ is just a formal infinitesimal which is "nilpotent" (i.e., $\epsilon^2 = 0$).

Then the data abstraction properties of the typed lambda calculus ensure that when you run the program with the dual number implementation of reals, you are returned a pair of the original value and its partial derivatives (i.e. the Jacobian matrix). Dan Piponi blogged about this over 15 years ago(!), but even though I read it at the time, the algorithm was so simple I failed to recognise its importance!

Reverse mode is more complicated -- it comes from the observation that in machine learning, you are generally optimising a function $R^n \to R$, where the result is some scoring function. Since n might be very large and the dimensionality of the result is just 1, you can achieve superior computational complexity by computing the transpose of the Jacobian rather than the Jacobian itself.

The Programming Languages Enthusiast | Developments in PL, and why they matter
"What is PL Research?" The Talk
7 years ago

pigworker in a space (Conor McBride)
Ming — a typecheckerchecker (episode 0)
7 years ago

Jeremy Siek
Reading list for getting started on Gradual Typing
7 years ago

Everything in Context
What we expect PhD students to know and learn
8 years ago

Why λ?
Defining backticks using "pipe" operations
8 years ago

Existential Type
Proofs by contradiction, versus contradiction proofs
9 years ago

Notes from a Medium-Sized Island
9 years ago

The Lab Lunch
Moving sites
9 years ago

Lazily Typed
Fresh Start
10 years ago

carlo angiuli (blog)
J is singleton contractibility plus transport, definitionally
11 years ago

Request for Logic
Life update, a TA blog
12 years ago

Proof Theory Abridged
Kripke models and the dialectica interpretation
13 years ago

The Paralyzing Paradoxes of Professor Polaro
reflecting and reifying the state monad
15 years ago

The Brown PLT Blog

Progress and Preservation

When you do so, the resulting syntactic program transformations all look like they are doing fancy continuation manipulations, and their semantic counterparts all look like manipulations of the dual space. So the syntatic continuation/double-negation operations are reflecting the fact that that finite-dimensional vector spaces are isomorphic to their double-duals (i.e., $V^{**} \simeq V$).

In addition to being really neat, this is actually a really nice problem to look at. People basically care about reverse mode AD because it goes fast, which means you get this lovely interplay between operational concerns and semantic models. A second bonus is that AD offers computer scientists a good reason to learn some differential geometry, which in turn is a good excuse to learn general relativity.

6? Incremental Computation

After all this talk of differentiation, you might wonder -- what about finite differences?

It turns out that they have a nice interpretation in lambda calculus, too! In 2015, Yufei Cai, Paolo Giarusso, Tillman Rendel and Klaus Ostermann introduced *the incremental lambda calculus*.

The basic idea behind this is that for each type X, you introduce a type of "changes" ΔX for it too. A morphism between $(X, \Delta X)$ and $(Y, \Delta Y)$ is a pair of functions $f : X \to Y$ and $df : X \to \Delta X \to \Delta Y$, where df is the incrementalization -- $df\ x\ dx$ tells you how the to change the output of f,x to account for the change dx to the input.

This looks a lot like what you get out of the tangent space functor, but it turns out that the incremental lambda calculus is *not* simply a derivative a la the differential lambda calculus, because the derivative of a function is unique, and these incrementalisations don't have to be. Michael Arntzenius and I used the ILC in our paper on extending seminaive evaluation from Datalog to a higher-order language, and it was critical for the efficiency of our result that we had this freedom.

Mario Picallo-Alvarez worked out answers to a lot of the semantics questions underlying this in his PhD thesis, *Change actions: from incremental computation to discrete derivatives*, showing how incremental lambda calculus arises as a generalisation of the models of the differential lambda calculus.

He noted that the "real" semantics for change types should look like a category, in which a type is a category where the objects are values, and a morphisms $\delta : v_1 \to v_2$ is evidence that v_1 can be changed into v_2. On the grounds that 2-categories are painful, Mario worked with change *monoids* (i.e., no typing, but yes to identities and composition), and the Giarusso-style change structures that Michael and I used retained the typed changes, but discarded all the categorical axioms. So there's more to be done here!

Posted by Neel Krishnaswami at 2:24 PM

1 comment:

sigfpe February 24, 2021 at 9:15 PM

The Brzozowski derivative doesn't satisfy the Leibniz rule because concatenation of strings isn't the product it interacts nicely with. When you replace concatenation with shuffle it all works out.

Reply

blog gallais

Bob Atkey's Blog

I use MathJax for mathematical layout.