A Simply Typed $\lambda$-Calculus of Forward Automatic Differentiation

Oleksandr Manzyuk

National University of Ireland Maynooth
manzyuk@gmail.com
Everyone in this audience knows what the simply typed \( \lambda \)-calculus is, but the words “forward automatic differentiation” probably sound less familiar. Therefore, I’d like to begin by quickly introducing you to automatic differentiation, commonly abbreviated AD, and I’d like to motivate AD by contrasting it with two other techniques for programmatically computing derivatives of functions.
Numerical Differentiation

... *approximates* the derivative:

\[ f'(x) \approx \frac{f(x + h) - f(x)}{h} \]

for a small value of \( h \).

*How small?*

- Too small values of \( h \) lead to large rounding errors.
- Too large values of \( h \) make the approximation inaccurate.
First, there is numerical differentiation, which approximates the derivative of a function \( f \) by Newton’s difference quotient for a small value of \( h \). The choice of a suitable \( h \) is a non-trivial problem because of the intricacies of floating point arithmetic. If \( h \) is too small, you are going to subtract two nearly equal numbers, which may cause extreme loss of accuracy. In fact, due to rounding errors, the difference in the numerator is going to be zero if \( h \) is small enough. On the other hand, if \( h \) is not sufficiently small, then the difference quotient is a bad estimate on the derivative.
Symbolic Differentiation

... uses a collection of rules:

\[(f + g)'(x) = f'(x) + g'(x)\]
\[(f \cdot g)'(x) = f'(x) \cdot g(x) + f(x) \cdot g'(x)\]
\[(f \circ g)'(x) = f'(g(x)) \cdot g'(x)\]
\[\exp'(x) = \exp(x)\]
\[\log'(x) = 1/x\]
\[\sin'(x) = \cos(x)\]
\[\cos'(x) = -\sin(x)\]

...
Second, there is symbolic differentiation, which works by applying the rules for computing derivatives (Leibniz rule, chain rule etc.) and by using a table of derivatives of elementary functions.

Unlike numerical differentiation, symbolic differentiation is exact.
Symbolic differentiation suffers from the *loss of sharing*. For example, consider computing the derivative of $f = f_1 \cdot \ldots \cdot f_n$:

$$f'(x) = f'_1(x) \cdot f_2(x) \cdot \ldots \cdot f_n(x)$$
$$+ f_1(x) \cdot f'_2(x) \cdot \ldots \cdot f_n(x)$$
$$\ldots$$
$$+ f_1(x) \cdot f_2(x) \cdot \ldots \cdot f'_n(x)$$

If evaluating $f_i(x)$ or $f'_i(x)$ each cost 1 and the arithmetic operations are free, then $f(x)$ has a cost of $n$, whereas $f'(x)$ has a cost of $n^2$. 
Unfortunately, symbolic differentiation can be very inefficient because it *loses sharing*. What do we mean by this? Let us illustrate with an example.

Consider the problem of computing the derivative of a product of $n$ functions. Applying the Leibniz rule, we arrive at the expression for the derivative, which has size quadratic in $n$. Evaluating it naively would result in evaluating each $f_i(x)$ $n - 1$ times. If our cost model is that evaluating $f_i(x)$ or $f'_i(x)$ each cost 1 and the arithmetic operations are free, then $f(x)$ has a cost of $n$, whereas $f'(x)$ has a cost of $n^2$.

The problem here is that in the expression produced by symbolic differentiation sharing is *implicit* and is not taken advantage of when the expression is evaluated. There are ways to fix this problem, e.g., by performing common subexpression elimination to make sharing *explicit*. As we shall see, forward AD accomplishes this by a clever trick.
Automatic Differentiation

... simultaneously manipulates values and derivatives.

Unlike numerical and symbolic differentiation, AD is

- exact
  - no rounding errors
  - as accurate as symbolic differentiation
- efficient
  - only a constant factor overhead
  - a lot of work can be moved to compile time
Finally, there is automatic differentiation, which as we shall see shortly simultaneously manipulates values and derivatives, leading to more sharing of the different instances of the derivative of a given subexpression in the computation of the derivative of a bigger expression.

Unlike numerical differentiation, AD is exact: there are no rounding errors, and in fact the answer produced by AD coincides with that produced by symbolic differentiation. Unlike symbolic differentiation, AD is efficient: if offers strong complexity guarantees (in particular, evaluation of the derivative takes no more than a constant factor times as many operations as evaluation of the function). It is also worth pointing out that using sophisticated compilation techniques it is possible to move a lot of work from run time to compile time.

AD comes in several variations: forward, reverse, as well as mixtures thereof. We shall only focus on forward AD.
Forward AD: Idea

Overload the primitives to operate both on real numbers, $\mathbb{R}$, and on dual numbers, $\mathbb{R}[\varepsilon]/(\varepsilon^2)$:

\[
(a_1 + \varepsilon b_1) + (a_2 + \varepsilon b_2) \overset{\text{def}}{=} (a_1 + a_2) + \varepsilon (b_1 + b_2),
\]

\[
(a_1 + \varepsilon b_1) \cdot (a_2 + \varepsilon b_2) \overset{\text{def}}{=} (a_1 \cdot b_1) + \varepsilon (a_1 \cdot b_2 + a_2 \cdot b_1),
\]

\[
p(x + \varepsilon x') \overset{\text{def}}{=} p(x) + \varepsilon p'(x) \cdot x',
\]

where $p \in \{\sin, \cos, \exp, \ldots \}$.

For any function $f$ built out of the overloaded primitives holds

\[
f(x + \varepsilon x') = f(x) + \varepsilon f'(x) \cdot x',
\]

which gives a recipe for computing the derivative of $f$. 
Forward AD can be implemented in several different ways, but so called overloading approach is the easiest to explain. The idea is to overload the primitives to operate both on real numbers and on dual numbers. Each dual number can be thought of as a pair consisting of a primal value and its “infinitesimally small” perturbation. The extension of each function $p$ from the numeric basis is given by essentially the formal Taylor series of $p$ truncated at degree 1.

What is interesting about this extension is that the chain rule for derivatives becomes encoded in function composition, and as a consequence any function $f$ built out of the overloaded primitives satisfies the equation $f(x + \varepsilon x') = f(x) + \varepsilon f'(x) \cdot x'$, which suggests a recipe for computing the derivative of $f$: evaluate $f$ at the point $x + \varepsilon$ and take the perturbation part of the obtained dual number.
Forward AD: Example

• Let \( f = \lambda x. x^2 + 1 \) and \( x = 3 \). Then:

\[
\begin{align*}
  f(3 + \varepsilon) &= (\lambda x. x^2 + 1)(3 + \varepsilon) \\
  &= (3 + \varepsilon) \cdot (3 + \varepsilon) + 1 \\
  &= 10 + 6\varepsilon,
\end{align*}
\]

hence \( f'(3) = 6 \).

• The derivative of \( f = f_1 \cdot \ldots \cdot f_n \) at \( x \):

\[
\begin{align*}
  f(x + \varepsilon) &= f_1(x + \varepsilon) \cdot \ldots \cdot f_n(x + \varepsilon) \\
  &= (f_1(x) + \varepsilon f_1'(x)) \cdot \ldots \cdot (f_n(x) + \varepsilon f_n'(x))
\end{align*}
\]

If evaluating \( f_i(x) \) or \( f_i'(x) \) each cost 1 and the arithmetic operations are free, then evaluating \( f'(x) \) has a cost of \( 2n \).
For example, the derivative of $\lambda x. x^2 + 1$ at the point 3 is obtained by evaluating the expression $x^2 + 1$ at the point $3 + \varepsilon$ interpreting $+$ and $\cdot$ as addition and multiplication of dual numbers, respectively, and taking the coefficient in front of $\varepsilon$.

Let us also illustrate how forward AD fixes the efficiency problem of symbolic differentiation. Consider again the problem of computing the derivative of a product of $n$ functions. Computing $f(x + \varepsilon)$ requires computing each $f_i(x + \varepsilon) = f_i(x) + \varepsilon f'_i(x)$, which has a cost of 2, and multiplying $n$ dual numbers, which is free in our cost model. Therefore, the total cost of computing $f(x + \varepsilon)$ is $2n$, only a constant factor overhead compared to the cost of computing $f(x)$. 
Forward AD: More Conceptual View

- The space of dual numbers $\mathbb{R}[\varepsilon]/(\varepsilon^2)$ is isomorphic to the tangent bundle $T\mathbb{R}$ over $\mathbb{R}$.

- The operations $+$ and $\cdot$ in $\mathbb{R}[\varepsilon]/(\varepsilon^2)$ correspond exactly to the pushforwards $T(+)$ and $T(\cdot)$ of the operations $+$ and $\cdot$ in $\mathbb{R}$.

- Extending a function $f : \mathbb{R} \to \mathbb{R}$ to dual numbers is punning the function and its pushforward $Tf : T\mathbb{R} \to T\mathbb{R}$.

- Forward AD is about applying the *compositional* properties of the tangent bundle functor $T$ rather than its definition to compute the pushforward of a function $f$. 
Here is a more conceptual view on forward AD.

First, notice that the space of dual numbers is isomorphic to the tangent bundle over the real line. Furthermore, the arithmetic operations on dual numbers correspond exactly to the pushforwards of the arithmetic operations on real number.

If we interpret the set of dual numbers as the tangent bundle over the real line, then the extension of a function $f$ given by the truncated Taylor series is nothing but the pushforward of $f$. Overloading a function means punning the function and its pushforward.

From this viewpoint, forward AD is really about computing pushforwards of functions using the compositional properties of the tangent bundle functor rather than its definition. By “compositional properties” we mean the functoriality, preservation of products etc.
Functional Programming Language with AD

• Our ultimate goal is to design a functional programming language with support for AD (Pearlmutter-Siskind). (BTW, there are PhD/postdoc positions available; contact barak@cs.nuim.ie.)

• To lay down a theoretical foundation for such a language, we would like to incorporate the notion of pushforward into λ-calculus.

• Here we embark on this project by extending the simply typed λ-calculus with pushforward operators.
This work was motivated by our desire to design a functional programming language with support for AD, the project initiated by Barak Pearlmutter and Jeff Siskind. BTW, we have PhD/postdoc positions available on the project. If you’re interested, please contact Barak. A majority of AD systems are built on top of imperative programming languages (FORTRAN, C/C++), whereas the idea of AD is most naturally embodied in a functional programming language. Indeed, the differential operator is almost a paradigmatic example of a higher-order function. However, despite a huge body of research and proliferation of AD implementations, a clear semantics of AD in the presence of first-class functions is lacking. In order to address this issue and to lay down a theoretical foundation for a functional programming language with support for forward AD, I suggest extending the λ-calculus with “pushforward operators”. Here we consider the simplest case: extending the simply typed λ-calculus.
Recap: The Simply Typed $\lambda$-Calculus

$M, N, \ldots ::= x \mid \lambda x. M \mid MN$

$\sigma, \tau, \ldots ::= \alpha \mid \sigma \to \tau$

$\Gamma, \Delta, \ldots ::= \emptyset \mid \Gamma; x : \sigma$

\[
\Gamma, x : \sigma \vdash x : \sigma
\]

\[
\Gamma, x : \sigma \vdash M : \tau
\]

\[
\Gamma \vdash \lambda x. M : \sigma \to \tau
\]

\[
\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma
\]

\[
\Gamma \vdash MN : \tau
\]
Before describing the extension, I’d like to give a very brief recap of the simply typed $\lambda$-calculus, mainly to introduce some notation and to recall some facts.

We are going to denote terms of the simply typed $\lambda$-calculus by capital letters $M$, $N$ etc. They are defined by the standard grammar consisting of three productions: variables, abstractions, applications. We also assume we are given a denumerable set of atomic types $\alpha$, and if $\sigma$ and $\tau$ are types, then so is $\sigma \rightarrow \tau$. Typing contexts are defined in the standard way as finite sequences of variable-type pairs. The typing rules are the standard ones.
Recap: Cartesian Closed Categories

\[ \begin{array}{c}
\text{Z} \\
\downarrow^f \downarrow^g \\
X \times Y \\
\downarrow^\pi_1 \downarrow^\pi_2 \\
X \leftarrow X \times Y \rightarrow Y
\end{array} \]

\[ \text{ev} = \text{ev}_{X,Y} : (X \Rightarrow Y) \times X \rightarrow Y \]

\[ \begin{array}{c}
Z \times X \\
\downarrow^{\Lambda(f) \times \text{id}_X} \\
(X \Rightarrow Y) \times X \\
\downarrow^f \downarrow^{\text{ev}_{X,Y}} \\
Y
\end{array} \]
We will also need some notation from the theory of cartesian closed categories.

We denote by angle brackets the pairing operation. The pairing of two morphisms $f : Z \to X$ and $g : Z \to Y$ is a unique morphism $\langle f, g \rangle : Z \to X \times Y$ whose compositions with the two projections are $f$ and $g$, respectively.

We write $X \Rightarrow Y$ for the internal Hom-object in a cartesian closed category. The evaluation morphism is denoted by $\text{ev}$.

We denote by $\Lambda(f)$ the currying of a morphism $f : Z \times X \to Y$, which is a unique morphism from $Z$ to $X \Rightarrow Y$ making this diagram commute.
Recap: Interpretation

Let \( \mathcal{C} \) be a cartesian closed category. Define the interpretation of the simply typed \( \lambda \)-calculus in the category \( \mathcal{C} \):

- \(|\alpha| = A, A \in \text{Ob} \mathcal{C}, |\sigma \to \tau| = |\sigma| \Rightarrow |\tau|;\)
- \(|\emptyset| = 1, |\Gamma; x : \sigma| = |\Gamma| \times |\sigma|;\)
- The interpretation of a judgment \( \Gamma \vdash M : \sigma \) is a morphism \( |M^\sigma|_\Gamma = |M|_\Gamma : |\Gamma| \to |\sigma| \) defined inductively as follows:

\[
|x^\sigma|_{\Gamma; x:\sigma} = \pi_2,
\]

\[
|y^\tau|_{\Gamma; x:\sigma} = |y^\tau|_\Gamma \circ \pi_1, \quad x \neq y,
\]

\[
|(\lambda x. M)^{\sigma \to \tau}|_\Gamma = \Lambda(|M^\tau|_{\Gamma; x:\sigma}),
\]

\[
|(MN)^\tau|_\Gamma = \text{ev} \circ \langle |M^\sigma^{\to \tau}|_\Gamma, |N^\sigma|_\Gamma \rangle.
\]

So defined interpretation is sound: \(|(\lambda x. M)N|_\Gamma = |M[N/x]|_\Gamma\) if \( \Gamma, x : \sigma \vdash M : \tau \) and \( \Gamma \vdash N : \sigma \).
Recap: Interpretation

The simply typed $\lambda$-calculus can be interpreted in a cartesian closed category as follows. Pick some objects of the category as interpretations of atomic types, and recursively define the interpretation of $\sigma \rightarrow \tau$ to be the internal Hom-object $|\sigma| \Rightarrow |\tau|$. Contexts are interpreted in the standard way: the interpretation of a context is the product of the interpretations of the types of the variables occurring in the context. The interpretation of a judgment $\Gamma \vdash M : \sigma$ is a morphism from $|\Gamma|$ to $|\sigma|$ defined inductively by these equations.

So defined interpretation is sound, which essentially means that beta-reduction preserves meaning of terms.
A Hypothetical Simply Typed $\lambda$-Calculus

We want to design an extension of the simply typed $\lambda$-calculus with pushforward operators such that:

- the semantics of the extension is an extension of the semantics of the simply typed $\lambda$-calculus;

- the pushforward operator of the calculus is modeled by some kind of pushforward operator of the model.
A Hypothetical Simply Typed $\lambda$-Calculus

Our goal is to extend the simply typed $\lambda$-calculus with pushforward operators. To do this in a systematic and principled way, we begin by identifying what we want the semantics of the hypothetical calculus to be. We have two natural conditions.

The semantics should be an extension of the semantics of the simply typed $\lambda$-calculus.

Pushforward operators of the calculus should be modeled by pushforward operators of the model of some kind. That is, we need a notion of cartesian closed category with pushforward operators.

What options are available to us? “Tangent structures” of Cockett and Cruttwell look promising, but they only consider tangent structures in cartesian not-necessarily-closed categories.

As a temporary solution, we use differential $\lambda$-categories as models for our hypothetical calculus.
A *cartesian differential category* (Blute-Cockett-Seely, 2009) is a cartesian left-additive category equipped with an operator

\[ D : \mathbb{C}(X, Y) \rightarrow \mathbb{C}(X \times X, Y) \]

satisfying the following axioms:

**D1.** \( D(f + g) = D(f) + D(g) \) and \( D(0) = 0 \).

**D2.** \( D(f) \circ \langle h + k, v \rangle = D(f) \circ \langle h, v \rangle + D(f) \circ \langle k, v \rangle \) and \( D(f) \circ \langle 0, v \rangle = 0 \).

**D3.** \( D(\text{id}) = \pi_1, \ D(\pi_1) = \pi_1 \circ \pi_1, \ D(\pi_2) = \pi_2 \circ \pi_1 \).

**D4.** \( D(\langle f, g \rangle) = \langle D(f), D(g) \rangle \).

**D5.** \( D(f \circ g) = D(f) \circ \langle D(g), g \circ \pi_2 \rangle \).

**D6.** \( D(D(f)) \circ \langle \langle g, 0 \rangle, \langle h, k \rangle \rangle = D(f) \circ \langle g, k \rangle \).

**D7.** \( D(D(f)) \circ \langle \langle 0, h \rangle, \langle g, k \rangle \rangle = D(D(f)) \circ \langle \langle 0, g \rangle, \langle h, k \rangle \rangle \).
First, I should explain what a differential λ-category is.

A differential λ-category is a cartesian differential category that is also closed and in which the closed structure is compatible with the differential structure.

Cartesian differential categories were introduced by Blute, Cockett, and Seely. A cartesian differential category is a left-additive cartesian category equipped with a differential operator $D$ satisfying axioms D1–D7.

Some intuition for the axioms: D1 says $D$ is additive; D2 that $D(f)$ is additive in its first coordinate; D3 and D4 assert that $D$ is compatible with the product structure, and D5 is the chain rule. D6 and D7 are harder to explain, but D6 means essentially that $D(f)$ is linear in its first variable and D7 is essentially independence of order of partial differentiation.
The category of smooth maps:

- objects are natural numbers;
- morphisms $m \to n$ are smooth maps $\mathbb{R}^m \to \mathbb{R}^n$;
- the operator $D$ takes an $f : \mathbb{R}^m \to \mathbb{R}^n$ and produces a $D(f) : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^n$ given by $D(f)(x', x) = J_f(x) \cdot x'$, where $J_f(x)$ is the Jacobian of $f$ at the point $x$. 
The paradigmatic example of a cartesian differential category is the category of smooth maps, whose objects are natural numbers and morphisms from $m$ to $n$ are smooth maps from $\mathbb{R}^m$ to $\mathbb{R}^n$. The operator $D$ takes a map $f$ from $\mathbb{R}^n$ to $\mathbb{R}^n$ and produces a $D(f)$ given by $D(f)(x', x) = J_f(x) \cdot x'$. Here $J_f(x)$ denotes the Jacobian of $f$ at the point $x$. 
A differential λ-category (Bucciarelli-Ehrhard-Manzonetto, 2010) is a cartesian closed differential category such that for each pair of objects \( X, Y \) the following diagram commutes:

\[
\begin{array}{ccc}
(((X \Rightarrow Y) \times (X \Rightarrow Y)) \times X & \xrightarrow{\pi_1 \times \text{id}_X} & (X \Rightarrow Y) \times X \\
\langle \pi_1 \times 0, \pi_2 \times \text{id}_X \rangle & \downarrow & \\
(((X \Rightarrow Y) \times X) \times ((X \Rightarrow Y) \times X)) & \xrightarrow{D(\text{ev})} & Y \\
\end{array}
\]

A morphism \( f : X \rightarrow Y \) is linear is \( D(f) = f \circ \pi_1 \). One can say that in a differential λ-category \( \text{ev} \) is “linear in the first variable”.

The category of convenient vector spaces and smooth maps is an example of differential λ-category.
Differential $\lambda$-categories were introduced by Bucciarelli, Ehrhard, and Manzonetto as categorical models for the differential $\lambda$-calculus of Ehrhard and Regnier.

A differential category is a cartesian closed differential category in which the following compatibility condition between the closed structure and the differential structure holds.

A morphism $f$ is called linear if its derivative is obtained by precomposing $f$ with the projection onto the first coordinate. One can express the compatibility condition by saying that the evaluation morphism is linear in the first variable.

Buccirarely et al. gave combinatorial examples of differential $\lambda$-categories. For our purposes, the example of convenient vector spaces and smooth maps is more intuitive.
Tangent Bundles in Differential $\lambda$-Categories

Let $C$ be a differential $\lambda$-category. The tangent bundle functor $T : C \to C$ is defined by $TX = X \times X$, $Tf = \langle D(f), f \circ \pi_2 \rangle$.

- $T$ is part of a monad $(T, \mu, \eta)$:
  \[
  \eta(x) = (0, x), \quad \mu((w, v), (u, x)) = (v + u, x).
  \]

- $t = t_{X,Y} : X \times TY \to T(X \times Y)$ given by
  \[
  t_{X,Y}(x, (y', y)) = ((0, y'), (x, y))
  \]

makes $(T, \mu, \eta)$ into a strong commutative monad.

- $T$ is a monoidal functor and hence admits an enrichment
  \[
  T = T_{X,Y} : X \Rightarrow Y \Rightarrow TX \Rightarrow TY.
  \]

The morphism $T$ is linear.
The differential operator $D$ allows us to replicate the construction of the tangent bundle of a smooth manifold from differential geometry in any cartesian differential category. Let $C$ be a cartesian differential category, and in fact a differential $\lambda$-category. The tangent bundle functor $T$ from $C$ to itself is defined by these formulas. This functor enjoys a number of nice properties:

- It is part of a monad (a fact that is somehow overlooked by the classical differential geometry). The unit is the zero section, and the multiplication is given by this formula. To simplify the notation, we write equations as if in the category of sets.

- The monad is strong commutative, with the right tensorial strength given by this formula.

- As a consequence, $T$ is a monoidal functor and hence admits a canonical enrichment $T$. We prove that $T$ is linear.
The Perturbative $\lambda$-Calculus

\[ M, N, \ldots ::= \cdots | 0 | M + N | \nu_k M | \pi_k M | T M \]

\[ \sigma, \tau, \ldots ::= \cdots | \sigma \times \tau \]

\[ \langle M, N \rangle \overset{\text{def}}{=} \nu_1 M + \nu_2 N \]

\[ \Gamma \vdash M : \sigma \quad \Gamma \vdash N : \sigma \]
\[ \Gamma \vdash 0 : \sigma \quad \Gamma \vdash M + N : \sigma \]

\[ \Gamma \vdash M : \sigma_k \quad \Gamma \vdash \pi_k M : \sigma_k \]
\[ \Gamma \vdash \nu_k M : \sigma_1 \times \sigma_2 \quad \Gamma \vdash M : \sigma_1 \times \sigma_2 \]

\[ \Gamma \vdash M : \sigma \rightarrow \tau \]
\[ \Gamma \vdash T M : T \sigma \rightarrow T \tau \]

\[ T \sigma \overset{\text{def}}{=} \sigma \times \sigma, \quad \left| (T M)^{T \sigma \rightarrow T \tau} \right|_\Gamma \overset{\text{def}}{=} T \circ \left| M^{\sigma \rightarrow \tau} \right|_\Gamma \]
We now want to translate the tangent bundle functor back to the \( \lambda \)-calculus as a syntactic operation. To this end, we extend the simply typed \( \lambda \)-calculus with new syntactic form: zero, sum of terms, injection of a term into the \( k \)th factor of a product, projection of a term onto the \( k \)th coordinate, and pushforward of a term. Sums reflect the left-additivity of differential \( \lambda \)-categories. Injections and projections (rather than pair constructors and projections) are introduced to syntactically capture additivity of pairs. Pairs are introduced as syntactic sugar. Types are extended with product types. New syntactic forms require new typing rules. The typing rules for sums, injections, and projections are the standard ones, and the typing rule for \( T \) matches our intended interpretation of \( T \) as the pushforward. On types, \( T \) is defined by \( T\,\sigma = \sigma \times \sigma \), which is motivated by our intention to interpret \( T\,\sigma \) as \( T|\sigma| \). The semantics of sums, injections, and projections is obvious. We postulate that the semantics of term \( T\,M \) should be \( T \) composed with the semantics of \( M \). This is our semantic equation.
The Reduction Rule for $T$

- We want to have a reduction rule of the form
  \[ T(\lambda x. M) \rightarrow \lambda x. T_x M, \]
  where $T_x M$ is to be defined. We want the definition of the transform $T_x$ to be compositional.

- We want to preserve the soundness of the semantics:
  \[ \left| (T(\lambda x. M))^\sigma \rightarrow^T \tau \right|_\Gamma = \left| (\lambda x. T_x M)^\sigma \rightarrow^T \tau \right|_\Gamma. \]

  From this equation, one can compute:
  \[ \left| (T_x M)^\tau \right|_{\Gamma; x:T \sigma} = T\left(\left| M^\tau \right|_{\Gamma; x:T \sigma}\right) \circ t. \]

- We can solve the above equation for $T_x M$, producing a set of equations defining $T_x$ recursively!
To make the obtained \( \lambda \)-calculus useful, in addition to beta-reductions for abstractions and products, we also need a reduction rule for \( T \). We are looking for a reduction rule of the form 
\[ T(\lambda x. M) \rightarrow \lambda x. T_x M, \]
where \( T_x M \) is a term to be defined. We want the definition of \( T_x \) to be compositional.

We also want to preserve the soundness of the semantics, so that the interpretation of the left hand side of the rule must be equal to the interpretation of the right hand side. From this condition, the interpretation of \( T_x M \) can be computed. We can then solve the obtained equation for \( T_x M \) by induction. This produces a set of equations defining \( T_x \) recursively.
The Definition of $T_x$

$$T_x y = \begin{cases} 
  x & \text{if } x = y \\
  \iota_2 y & \text{otherwise}
\end{cases}$$

$$T_x (\lambda y. M) = \langle \lambda y. \pi_1(T_x M), \lambda y. \pi_2(T_x M) \rangle$$

$$T_x (MN) = (T_x M) \diamond (T_x N)$$

$$T_x (\pi M) = \langle T (\pi_1(T_x M)), T (\pi_2(T_x M)) \rangle$$

$$T_x 0 = 0$$

$$T_x (M + N) = T_x M + T_x N$$

$$T_x (\iota_k M) = \langle \iota_k(\pi_1(T_x M)), \iota_k(\pi_2(T_x M)) \rangle$$

$$T_x (\pi_k M) = \langle \pi_k(\pi_1(T_x M)), \pi_k(\pi_2(T_x M)) \rangle$$

$$M \diamond N \overset{\text{def}}{=} \langle (\pi_1 M)(\pi_2 N) + \pi_1((T (\pi_2 M)) N), (\pi_2 M)(\pi_2 N) \rangle$$
Here is a full set of equations. The definition of $T_x$ is probably the most interesting contribution of the paper. This definition makes the interpretation of our $\lambda$-calculus in differential $\lambda$-categories sound, essentially by design. Note that this is not the way things are presented in the paper. There, the definition of $T_x$ is taken from thin air and is proved to be sound.
Conclusions

- We have extended the simply typed $\lambda$-calculus with the pushforward operator.
- We have derived a reduction rule for the pushforward operator from semantic considerations.
- The interpretation of the perturbative $\lambda$-calculus is sound with respect to this reduction rule (by design!) as well as other, more standard reduction rules.
- We conjecture that this reduction rule can be made part of a confluent and strongly normalizing rewriting system.
Conclusions

Let us summarize what we have done.

We have extended the simply typed $\lambda$-calculus with the pushforward operator. This extension was motivated by our desire to incorporate the notion of pushforward into a functional programming language. We have used the theory of tangent bundles in differential $\lambda$-categories to drive the design of the calculus.

We have derived a reduction rule for the pushforward operator from semantic consideration. As a consequence, the obtained $\lambda$-calculus can be soundly modeled by differential $\lambda$-categories.

So, what's next? We conjecture that this reduction rule together with other reduction rules (beta-reduction for abstractions and products, maybe eta-reduction for products) is confluent and strongly normalizing.