



Correctness of Automatic Differentiation via Diffeologies and Categorical Gluing

Mathieu Huot ^{1*}, Sam Staton^{1*}, and Matthijs Vákár^{2*}

¹ University of Oxford, UK

² Utrecht University, The Netherlands

*Equal contribution mathieu.huot@stx.ox.ac.uk

Abstract. We present semantic correctness proofs of Automatic Differentiation (AD). We consider a forward-mode AD method on a higher order language with algebraic data types, and we characterise it as the unique structure preserving macro given a choice of derivatives for basic operations. We describe a rich semantics for differentiable programming, based on diffeological spaces. We show that it interprets our language, and we phrase what it means for the AD method to be correct with respect to this semantics. We show that our characterisation of AD gives rise to an elegant semantic proof of its correctness based on a gluing construction on diffeological spaces. We explain how this is, in essence, a logical relations argument. Finally, we sketch how the analysis extends to other AD methods by considering a continuation-based method.

1 Introduction

Automatic differentiation (AD), loosely speaking, is the process of taking a program describing a function, and building the derivative of that function by applying the chain rule across the program code. As gradients play a central role in many aspects of machine learning, so too do automatic differentiation systems such as TensorFlow [1] or Stan [6].

Differentiation has a well developed mathematical theory in terms of differential geometry. The aim of this paper is to formalize this connection between differential geometry and the syntactic operations of AD. In this way we achieve two things: (1) a compositional, denotational understanding of differentiable programming and AD; (2) an explanation of the correctness of AD.

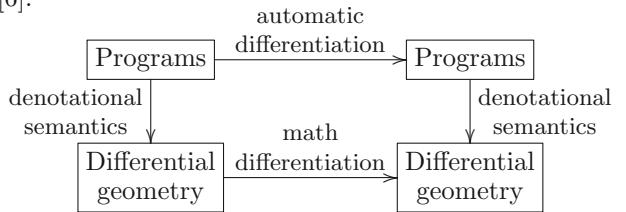


Fig. 1. Overview of semantics/correctness of AD.

This intuitive correspondence (summarized in Fig. 1) is in fact rather complicated. In this paper we focus on resolving the following problem: higher order functions play a key role in programming, and yet they have no counterpart in traditional differential geometry. Moreover, we resolve this problem while retaining the compositionality of denotational semantics.

Higher order functions and differentiation. A major application of higher order functions is to support disciplined code reuse. Code reuse is particularly acute in machine learning. For example, a multi-layer neural network might be built of millions of near-identical neurons, as follows.

$$\text{neuron}_n : (\mathbf{real}^n * (\mathbf{real}^n * \mathbf{real})) \rightarrow \mathbf{real}$$

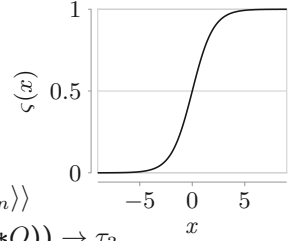
$$\text{neuron}_n \stackrel{\text{def}}{=} \lambda \langle x, \langle w, b \rangle \rangle. \varsigma(w \cdot x + b)$$

$$\text{layer}_n : ((\tau_1 * P) \rightarrow \tau_2) \rightarrow (\tau_1 * P^n) \rightarrow \tau_2^n$$

$$\text{layer}_n \stackrel{\text{def}}{=} \lambda f. \lambda \langle x, \langle p_1, \dots, p_n \rangle \rangle. \langle f \langle x, p_1 \rangle, \dots, f \langle x, p_n \rangle \rangle$$

$$\text{comp} : (((\tau_1 * P) \rightarrow \tau_2) * ((\tau_2 * Q) \rightarrow \tau_3)) \rightarrow (\tau_1 * (P * Q)) \rightarrow \tau_3$$

$$\text{comp} \stackrel{\text{def}}{=} \lambda \langle f, g \rangle. \lambda \langle x, \langle p, q \rangle \rangle. g \langle f \langle x, p \rangle, q \rangle$$



(Here $\varsigma(x) \stackrel{\text{def}}{=} \frac{1}{1+e^{-x}}$ is the sigmoid function, as illustrated.) We can use these functions to build a network as follows (see also Fig. 2):

$$\text{comp} \langle \text{layer}_m(\text{neuron}_k), \text{comp} \langle \text{layer}_n(\text{neuron}_m), \text{neuron}_n \rangle \rangle : (\mathbf{real}^k * P) \rightarrow \mathbf{real} \quad (1)$$

Here $P \cong \mathbf{real}^p$ with $p = (m(k+1) + n(m+1) + n+1)$. This program (1) describes a smooth (infinitely differentiable) function. The goal of automatic differentiation is to find its derivative.

If we β -reduce all the λ 's, we end up with a very long function expression just built from the sigmoid function and linear algebra. We can then find a program for calculating its derivative by applying the chain rule. However, automatic differentiation can also be expressed without first β -reducing, in a compositional way, by explaining how higher order functions like (layer) and (comp) propagate derivatives.

This paper is a semantic analysis of this compositional approach.

The general idea of denotational semantics is to interpret types as spaces and programs as functions between the spaces. In this paper, we propose to use diffeological spaces and smooth functions [32, 16] to this end. These satisfy the following three desiderata:

- \mathbb{R} is a space, and the smooth functions $\mathbb{R} \rightarrow \mathbb{R}$ are exactly the functions that are infinitely differentiable;
- The set of smooth functions $X \rightarrow Y$ between spaces again forms a space, so we can interpret function types.
- The disjoint union of a sequence of spaces again forms a space, so we can interpret variant types and inductive types.

We emphasise that the most standard formulation of differential geometry, using manifolds, does not support spaces of functions. Diffeological spaces seem to us the simplest notion of space that satisfies these conditions, but there are other

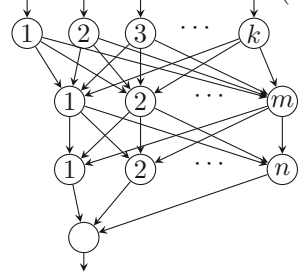


Fig. 2. The network in (1) with k inputs and two hidden layers.

candidates [3, 33]. A diffeological space is in particular a set X equipped with a chosen set of curves $C_X \subseteq X^{\mathbb{R}}$ and a smooth map $f : X \rightarrow Y$ must be such that if $\gamma \in C_X$ then $\gamma; f \in C_Y$. This is reminiscent of the method of logical relations.

From smoothness to automatic derivatives at higher types. Our denotational semantics in diffeological spaces guarantees that all definable functions are smooth. But we need more than just to know that a definable function happens to have a mathematical derivative: we need to be able to find that derivative.

In this paper we focus on a simple, forward mode automatic differentiation method, which is a macro translation on syntax (called \vec{D} in §2). We are able to show that it is correct, using our denotational semantics.

Here there is one subtle point that is central to our development. Although differential geometry provides established derivatives for first order functions (such as neuron above), there is no canonical notion of derivative for higher order functions (such as layer and comp) in the theory of diffeological spaces (e.g. [7]). We propose a new way to resolve this, by interpreting types as triples (X, X', S) where, intuitively, X is a space of inhabitants of the type, X' is a space serving as a chosen bundle of tangents over X , and $S \subseteq X^{\mathbb{R}} \times X'^{\mathbb{R}}$ is a binary relation between curves, informally relating curves in X with their tangent curves in X' . This new model gives a denotational semantics for automatic differentiation.

In §3 we boil this new approach down to a straightforward and elementary logical relations argument for the correctness of automatic differentiation. The approach is explained in detail in §5.

Related work and context. AD has a long history and has many implementations. AD was perhaps first phrased in a functional setting in [26], and there are now a number of teams working on AD in the functional setting (e.g. [34, 31, 12]), some providing efficient implementations. Although that work does not involve formal semantics, it is inspired by intuitions from differential geometry and category theory.

This paper adds to a very recent body of work on verified automatic differentiation. Much of this is concurrent with and independent from the work in this article. In the first order setting, there are recent accounts based on denotational semantics in manifolds [13] and based on synthetic differential geometry [9], as well as work making a categorical abstraction [8] and work connecting operational semantics with denotational semantics [2, 28]. Recently there has also been significant progress at higher types. The work of Brunel et al. gives formal correctness proofs for reverse-mode derivatives on computation graphs [5]. The work of Barthe et al. [4] provides a general discussion of some new syntactic logical relations arguments including one very similar to our syntactic proof of Theorem 1. We understand that the authors of [9] are working on higher types.

The differential λ -calculus [11] is related to AD, and explicit connections are made in [22, 23]. One difference is that the differential λ -calculus allows addition of terms at all types, and hence vector space models are suitable, but this appears peculiar with the variant and inductive types that we consider here.

Finally we emphasise that we have chosen the neural network (1) as our running example mainly for its simplicity. There are many other examples of AD

outside the neural networks literature: AD is useful whenever derivatives need to be calculated on high dimensional spaces. This includes optimization problems more generally, where the derivative is passed to a gradient descent method (e.g. [30, 18, 29, 19, 10, 21]). Other applications of AD are in advanced *integration* methods, since derivatives play a role in Hamiltonian Monte Carlo [25, 14] and variational inference [20].

Summary of contributions. We have provided a semantic analysis of automatic differentiation. Our syntactic starting point is a well-known forward-mode AD macro on a typed higher order language (e.g. [31, 34]). We recall this in §2 for function types, and in §4 we extend it to inductive types and variants. The main contributions of this paper are as follows.

- We give a denotational semantics for the language in diffeological spaces, showing that every definable expression is smooth (§3).
- We show correctness of the AD macro by a logical relations argument (Th. 1).
- We give a categorical analysis of this correctness argument with two parts: canonicity of the macro in terms of syntactic categories, and a new notion of glued space that abstracts the logical relation (§5).
- We then use this analysis to state and prove a correctness argument at all first order types (Th. 2).
- We show that our method is not specific to one particular AD macro, by also considering a continuation-based AD method (§6).

2 A simple forward-mode AD translation

Rudiments of differentiation and dual numbers. Recall that the derivative of a function $f : \mathbb{R} \rightarrow \mathbb{R}$, if it exists, is a function $\nabla f : \mathbb{R} \rightarrow \mathbb{R}$ such that $\nabla f(x_0) = \frac{df(x)}{dx}(x_0)$ is the gradient of f at x_0 .

To find ∇f in a compositional way, two generalizations are reasonable:

- We need both f and ∇f when calculating $\nabla(f;g)$ of a composition $f;g$, using the chain rule, so we are really interested in the pair $(f, \nabla f) : \mathbb{R} \rightarrow \mathbb{R} \times \mathbb{R}$;
- In building f we will need to consider functions of multiple arguments, such as $+: \mathbb{R}^2 \rightarrow \mathbb{R}$, and these functions should propagate derivatives.

Thus we are more generally interested in transforming a function $g : \mathbb{R}^n \rightarrow \mathbb{R}$ into a function $h : (\mathbb{R} \times \mathbb{R})^n \rightarrow \mathbb{R} \times \mathbb{R}$ in such a way that for any $f_1 \dots f_n : \mathbb{R} \rightarrow \mathbb{R}$,

$$(f_1, \nabla f_1, \dots, f_n, \nabla f_n); h = ((f_1, \dots, f_n); g, \nabla((f_1, \dots, f_n); g)). \quad (2)$$

An intuition for h is often given in terms of dual numbers. The transformed function operates on pairs of numbers, (x, x') , and it is common to think of such a pair as $x + x'\epsilon$ for an ‘infinitesimal’ ϵ . But while this is a helpful intuition, the formalization of infinitesimals can be intricate, and the development in this paper is focussed on the elementary formulation in (2).

The reader may also notice that h encodes all the partial derivatives of g . For example, if $g : \mathbb{R}^2 \rightarrow \mathbb{R}$, then with $f_1(x) \stackrel{\text{def}}{=} x$ and $f_2(x) \stackrel{\text{def}}{=} x_2$, by applying (2) to x_1 we obtain $h(x_1, 1, x_2, 0) = (g(x_1, x_2), \frac{\partial g(x, x_2)}{\partial x}(x_1))$ and similarly

$h(x_1, 0, x_2, 1) = (g(x_1, x_2), \frac{\partial g(x_1, x)}{\partial x}(x_2))$. And conversely, if g is differentiable in each argument, then a unique h satisfying (2) can be found by taking linear combinations of partial derivatives:

$$h(x_1, x'_1, x_2, x'_2) = (g(x_1, x_2), x'_1 \cdot \frac{\partial g(x, x_2)}{\partial x}(x_1) + x'_2 \cdot \frac{\partial g(x_1, x)}{\partial x}(x_2)).$$

In summary, the idea of differentiation with dual numbers is to transform a differentiable function $g : \mathbb{R}^n \rightarrow \mathbb{R}$ to a function $h : \mathbb{R}^{2n} \rightarrow \mathbb{R}^2$ which captures g and all its partial derivatives. We packaged this up in (2) as a sort-of invariant which is useful for building derivatives of compound functions $\mathbb{R} \rightarrow \mathbb{R}$ in a compositional way. The idea of forward mode automatic differentiation is to perform this transformation at the source code level.

A simple language of smooth functions. We consider a standard higher order typed language with a first order type **real** of real numbers. The types (τ, σ) and terms (t, s) are as follows.

$\tau, \sigma, \rho ::=$	types	$(\tau_1 * \dots * \tau_n)$	finite product
real	real numbers	$\tau \rightarrow \sigma$	function
$t, s, r ::=$	terms		
x	variable		
\underline{c} $t + s$ $t * s$ $\varsigma(t)$	operations/constants		
$\langle t_1, \dots, t_n \rangle$ case t of $\langle x_1, \dots, x_n \rangle \rightarrow s$	tuples/pattern matching		
$\lambda x. t$ $t s$	function abstraction/app.		

The typing rules are in Figure 3. We have included a minimal set of operations for the sake of illustration, but it is not difficult to add further operations. We add some simple syntactic sugar $t - u \stackrel{\text{def}}{=} t + (-1) * u$. We intend ς to stand for the sigmoid function, $\varsigma(x) \stackrel{\text{def}}{=} \frac{1}{1+e^{-x}}$. We further include syntactic sugar **let** $x = t$ **in** s for $(\lambda x. s) t$ and $\lambda \langle x_1, \dots, x_n \rangle. t$ for $\lambda x. \text{case } x \text{ of } \langle x_1, \dots, x_n \rangle \rightarrow t$.

Syntactic automatic differentiation: a functorial macro. The aim of forward mode AD is to find the dual numbers representation of a function by syntactic manipulations. For our simple language, we implement this as the following inductively defined macro $\vec{\mathcal{D}}$ on both types and terms (see also [34, 31]):

$$\begin{aligned} \vec{\mathcal{D}}(\text{real}) &\stackrel{\text{def}}{=} (\text{real} * \text{real}) & \vec{\mathcal{D}}(\tau \rightarrow \sigma) &\stackrel{\text{def}}{=} \vec{\mathcal{D}}(\tau) \rightarrow \vec{\mathcal{D}}(\sigma) \\ \vec{\mathcal{D}}((\tau_1 * \dots * \tau_n)) &\stackrel{\text{def}}{=} (\vec{\mathcal{D}}(\tau_1) * \dots * \vec{\mathcal{D}}(\tau_n)) \end{aligned}$$

$\frac{}{\Gamma \vdash \underline{c} : \text{real}} (c \in \mathbb{R})$	$\frac{\Gamma \vdash t : \text{real} \quad \Gamma \vdash s : \text{real}}{\Gamma \vdash t + s : \text{real}}$	$\frac{\Gamma \vdash t : \text{real} \quad \Gamma \vdash s : \text{real}}{\Gamma \vdash t * s : \text{real}}$	$\frac{\Gamma \vdash t : \text{real}}{\Gamma \vdash \varsigma(t) : \text{real}}$
$\frac{\Gamma \vdash t_1 : \tau_1 \quad \dots \quad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash \langle t_1, \dots, t_n \rangle : (\tau_1 * \dots * \tau_n)}$	$\frac{\Gamma \vdash t : (\sigma_1 * \dots * \sigma_n) \quad \Gamma, x_1 : \sigma_1, \dots, x_n : \sigma_n \vdash s : \tau}{\Gamma \vdash \text{case } t \text{ of } \langle x_1, \dots, x_n \rangle \rightarrow s : \tau}$		
$\frac{}{\Gamma \vdash x : \tau} ((x : \tau) \in \Gamma)$	$\frac{\Gamma, x : \tau \vdash t : \sigma}{\Gamma \vdash \lambda x : \tau. t : \tau \rightarrow \sigma}$	$\frac{\Gamma \vdash t : \sigma \rightarrow \tau \quad \Gamma \vdash s : \sigma}{\Gamma \vdash t s : \tau}$	

Fig. 3. Typing rules for the simple language.

$$\begin{aligned}
\vec{D}(x) &\stackrel{\text{def}}{=} x & \vec{D}(\underline{c}) &\stackrel{\text{def}}{=} \langle \underline{c}, 0 \rangle \\
\vec{D}(t + s) &\stackrel{\text{def}}{=} \mathbf{case} \vec{D}(t) \mathbf{of} \langle x, x' \rangle \rightarrow \mathbf{case} \vec{D}(s) \mathbf{of} \langle y, y' \rangle \rightarrow \langle x + y, x' + y' \rangle \\
\vec{D}(t * s) &\stackrel{\text{def}}{=} \mathbf{case} \vec{D}(t) \mathbf{of} \langle x, x' \rangle \rightarrow \mathbf{case} \vec{D}(s) \mathbf{of} \langle y, y' \rangle \rightarrow \langle x * y, x * y' + x' * y \rangle \\
\vec{D}(\varsigma(t)) &\stackrel{\text{def}}{=} \mathbf{case} \vec{D}(t) \mathbf{of} \langle x, x' \rangle \rightarrow \mathbf{let} y = \varsigma(x) \mathbf{in} \langle y, x' * y * (1 - y) \rangle \\
\vec{D}(\lambda x. t) &\stackrel{\text{def}}{=} \lambda x. \vec{D}(t) & \vec{D}(t s) &\stackrel{\text{def}}{=} \vec{D}(t) \vec{D}(s) & \vec{D}(\langle t_1, \dots, t_n \rangle) &\stackrel{\text{def}}{=} \langle \vec{D}(t_1), \dots, \vec{D}(t_n) \rangle \\
\vec{D}(\mathbf{case} t \mathbf{of} \langle x_1, \dots, x_n \rangle \rightarrow s) &\stackrel{\text{def}}{=} \mathbf{case} \vec{D}(t) \mathbf{of} \langle x_1, \dots, x_n \rangle \rightarrow \vec{D}(s)
\end{aligned}$$

We extend \vec{D} to contexts: $\vec{D}(\{x_1:\tau_1, \dots, x_n:\tau_n\}) \stackrel{\text{def}}{=} \{x_1:\vec{D}(\tau_1), \dots, x_n:\vec{D}(\tau_n)\}$. This turns \vec{D} into a well-typed, functorial macro in the following sense.

Lemma 1 (Functorial macro). *If $\Gamma \vdash t : \tau$ then $\vec{D}(\Gamma) \vdash \vec{D}(t) : \vec{D}(\tau)$.*

If $\Gamma, x : \sigma \vdash t : \tau$ and $\Gamma \vdash s : \sigma$ then $\vec{D}(\Gamma) \vdash \vec{D}(t[s/x]) = \vec{D}(t)[\vec{D}(s)/x]$.

Example 1 (Inner products). Let us write τ^n for the n -fold product $(\tau * \dots * \tau)$. Then, given $\Gamma \vdash t, s : \mathbf{real}^n$ we can define their inner product

$$\begin{aligned}
\Gamma \vdash t \cdot_n s &\stackrel{\text{def}}{=} \mathbf{case} t \mathbf{of} \langle z_1, \dots, z_n \rangle \rightarrow \\
&\quad \mathbf{case} s \mathbf{of} \langle y_1, \dots, y_n \rangle \rightarrow z_1 * y_1 + \dots + z_n * y_n : \mathbf{real}
\end{aligned}$$

To illustrate the calculation of \vec{D} , let us expand (and β -reduce) $\vec{D}(t \cdot_2 s)$:

$$\begin{aligned}
&\mathbf{case} \vec{D}(t) \mathbf{of} \langle z_{1,1}, z_{2,1} \rangle \rightarrow \mathbf{case} \vec{D}(s) \mathbf{of} \langle y_{1,1}, y_{2,1} \rangle \rightarrow \mathbf{case} z_{1,1} \mathbf{of} \langle z_{1,1,1}, z_{1,1,2} \rangle \rightarrow \\
&\mathbf{case} y_{1,1} \mathbf{of} \langle y_{1,1,1}, y_{1,1,2} \rangle \rightarrow \mathbf{case} z_{2,1} \mathbf{of} \langle z_{2,1,1}, z_{2,1,2} \rangle \rightarrow \mathbf{case} y_{2,1} \mathbf{of} \langle y_{2,1,1}, y_{2,1,2} \rangle \rightarrow \\
&\langle z_{1,1,1} * y_{1,1,1} + z_{2,1,1} * y_{2,1,1}, z_{1,1,2} * y_{1,1,2} + z_{2,1,2} * y_{2,1,2} + z_{1,1,1} * y_{2,1,2} + z_{2,1,1} * y_{1,1,2} \rangle
\end{aligned}$$

Example 2 (Neural networks). In our introduction (1), we provided a program in our language to build a neural network out of expressions `neuron`, `layer`, `comp`; this program makes use of the inner product of Ex. 1. We can similarly calculate \vec{D} of such deep neural nets by mechanically applying the macro.

3 Semantics of differentiation

Consider for a moment the first order fragment of the language in § 2, with only one type, **real**, and no λ 's or pairs. This has a simple semantics in the category of cartesian spaces and smooth maps. Indeed, a term $x_1 \dots x_n : \mathbf{real} \vdash t : \mathbf{real}$ has a natural reading as a function $\llbracket t \rrbracket : \mathbb{R}^n \rightarrow \mathbb{R}$ by interpreting our operation symbols by the well-known operations on $\mathbb{R}^n \rightarrow \mathbb{R}$ with the corresponding name. In fact, the functions that are definable in this first order fragment are smooth, which means that they are continuous, differentiable, and their derivatives are continuous, differentiable, and so on. Let us write **CartSp** for this category of cartesian spaces (\mathbb{R}^n for some n) and smooth functions.

The category **CartSp** has cartesian products, and so we can also interpret product types, tupling and pattern matching, giving us a useful syntax for constructing functions into and out of products of \mathbb{R} . For example, the interpretation of `(neuronn)` in (1) becomes

$$\mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \xrightarrow{\llbracket \cdot \rrbracket \times \text{id}_{\mathbb{R}}} \mathbb{R} \times \mathbb{R} \xrightarrow{\llbracket + \rrbracket} \mathbb{R} \xrightarrow{\llbracket \varsigma \rrbracket} \mathbb{R}.$$

where $\llbracket \cdot \rrbracket$, $\llbracket + \rrbracket$ and $\llbracket \varsigma \rrbracket$ are the usual inner product, addition and the sigmoid function on \mathbb{R} , respectively.

Inside this category, we can straightforwardly study the first order language without λ 's, and automatic differentiation. In fact, we can prove the following by plain induction on the syntax:

The interpretation of the (syntactic) forward AD $\vec{\mathcal{D}}(t)$ of a first-order term t equals the usual (semantic) derivative of the interpretation of t as a smooth function.

However, as is well known, the category **CartSp** does not support function spaces. To see this, notice that we have polynomial terms

$$x_1, \dots, x_d : \mathbf{real} \vdash \lambda y. \sum_{n=1}^d x_n y^n : \mathbf{real} \rightarrow \mathbf{real}$$

for each d , and so if we could interpret $(\mathbf{real} \rightarrow \mathbf{real})$ as a Euclidean space \mathbb{R}^p then, by interpreting these polynomial expressions, we would be able to find continuous injections $\mathbb{R}^d \rightarrow \mathbb{R}^p$ for every d , which is topologically impossible for any p , for example as a consequence of the Borsuk-Ulam theorem (see [15], Appx. A).

This means that we cannot interpret the functions (layer) and (comp) from (1) in **CartSp**, as they are higher order functions, even though they are very useful and innocent building blocks for differential programming! Clearly, we could define neural nets such as (1) directly as smooth functions without any higher order subcomponents, though that would quickly become cumbersome for deep networks. A problematic consequence of the lack of a semantics for higher order differential programs is that we have no obvious way of establishing compositional semantic correctness of $\vec{\mathcal{D}}$ for the given implementation of (1).

Diffeological spaces. This motivates us to turn to a more general notion of differential geometry for our semantics, based on *diffeological spaces* [16]. The key idea will be that a higher order function is called smooth if it sends smooth functions to smooth functions, meaning that we can never use it to build first order functions that are not smooth. For example, (comp) in (1) has this property.

Definition 1. A diffeological space (X, \mathcal{P}_X) consists of a set X together with, for each n and each open subset U of \mathbb{R}^n , a set $\mathcal{P}_X^U \subseteq [U \rightarrow X]$ of functions, called plots, such that

- all constant functions are plots;
- if $f : V \rightarrow U$ is a smooth function and $p \in \mathcal{P}_X^U$, then $f \circ p \in \mathcal{P}_X^V$;
- if $(p_i \in \mathcal{P}_X^{U_i})_{i \in I}$ is a compatible family of plots ($x \in U_i \cap U_j \Rightarrow p_i(x) = p_j(x)$) and $(U_i)_{i \in I}$ covers U , then the gluing $p : U \rightarrow X : x \in U_i \mapsto p_i(x)$ is a plot.

We call a function $f : X \rightarrow Y$ between diffeological spaces *smooth* if, for all plots $p \in \mathcal{P}_X^U$, we have that $p \circ f \in \mathcal{P}_Y^U$. We write **Diff** (X, Y) for the set of smooth maps from X to Y . Smooth functions compose, and so we have a category **Diff** of diffeological spaces and smooth functions.

A diffeological space is thus a set equipped with structure. Many constructions of sets carry over straightforwardly to diffeological spaces.

Example 3 (Cartesian diffeologies). Each open subset U of \mathbb{R}^n can be given the structure of a diffeological space by taking all the smooth functions $V \rightarrow U$

as \mathcal{P}_U^V . It is easily seen that smooth functions from $V \rightarrow U$ in the traditional sense coincide with smooth functions in the sense of diffeological spaces. Thus diffeological spaces have a profound relationship with ordinary calculus.

In categorical terms, this gives a full embedding of **CartSp** in **Diff**.

Example 4 (Product diffeologies). Given a family $(X_i)_{i \in I}$ of diffeological spaces, we can equip the product $\prod_{i \in I} X_i$ of sets with the *product diffeology* in which U -plots are precisely the functions of the form $(p_i)_{i \in I}$ for $p_i \in \mathcal{P}_{X_i}^U$.

This gives us the categorical product in **Diff**.

Example 5 (Functional diffeology). We can equip the set **Diff**(X, Y) of smooth functions between diffeological spaces with the *functional diffeology* in which U -plots consist of functions $f : U \rightarrow \mathbf{Diff}(X, Y)$ such that $(u, x) \mapsto f(u)(x)$ is an element of **Diff**($U \times X, Y$).

This specifies the categorical function object in **Diff**.

Semantics and correctness of AD. We can now give a denotational semantics to our language from § 2. We interpret each type τ as a set $\llbracket \tau \rrbracket$ equipped with the relevant diffeology, by induction on the structure of types:

$$\llbracket \mathbf{real} \rrbracket \stackrel{\text{def}}{=} \mathbb{R} \quad \llbracket (\tau_1 * \dots * \tau_n) \rrbracket \stackrel{\text{def}}{=} \prod_{i=1}^n \llbracket \tau_i \rrbracket \quad \llbracket \tau \rightarrow \sigma \rrbracket \stackrel{\text{def}}{=} \mathbf{Diff}(\llbracket \tau \rrbracket, \llbracket \sigma \rrbracket)$$

A context $\Gamma = (x_1 : \tau_1 \dots x_n : \tau_n)$ is interpreted as a diffeological space $\llbracket \Gamma \rrbracket \stackrel{\text{def}}{=} \prod_{i=1}^n \llbracket \tau_i \rrbracket$. Now well typed terms $\Gamma \vdash t : \tau$ are interpreted as smooth functions $\llbracket t \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket \tau \rrbracket$, giving a meaning for t for every valuation of the context. This is routinely defined by induction on the structure of typing derivations. Constants $\underline{c} : \mathbf{real}$ are interpreted as constant functions; and the first order operations $(+, *, \varsigma)$ are interpreted by composing with the corresponding functions, which are smooth. For example, $\llbracket \varsigma(t) \rrbracket(\rho) \stackrel{\text{def}}{=} \varsigma(\llbracket t \rrbracket(\rho))$, where $\rho \in \llbracket \Gamma \rrbracket$. Variables are interpreted as $\llbracket x_i \rrbracket(\rho) \stackrel{\text{def}}{=} \rho_i$. The remaining constructs are interpreted as follows, and it is straightforward to show that smoothness is preserved.

$$\begin{aligned} \llbracket \langle t_1, \dots, t_n \rangle \rrbracket(\rho) &\stackrel{\text{def}}{=} (\llbracket t_1 \rrbracket(\rho), \dots, \llbracket t_n \rrbracket(\rho)) & \llbracket \lambda x : \tau. t \rrbracket(\rho)(a) &\stackrel{\text{def}}{=} \llbracket t \rrbracket(\rho, a) \quad (a \in \llbracket \tau \rrbracket) \\ \llbracket \mathbf{case } t \mathbf{ of } \langle \dots \rangle \rightarrow s \rrbracket(\rho) &\stackrel{\text{def}}{=} \llbracket s \rrbracket(\rho, \llbracket t \rrbracket(\rho)) & \llbracket t s \rrbracket(\rho) &\stackrel{\text{def}}{=} \llbracket t \rrbracket(\rho)(\llbracket s \rrbracket(\rho)) \end{aligned}$$

Notice that a term $x_1 : \mathbf{real}, \dots, x_n : \mathbf{real} \vdash t : \mathbf{real}$ is interpreted as a smooth function $\llbracket t \rrbracket : \mathbb{R}^n \rightarrow \mathbb{R}$, even if t involves higher order functions (like (1)). Moreover the macro differentiation $\vec{\partial}(t)$ is a function $\llbracket \vec{\partial}(t) \rrbracket : (\mathbb{R} \times \mathbb{R})^n \rightarrow (\mathbb{R} \times \mathbb{R})$. This enables us to state a limited version of our main correctness theorem:

Theorem 1 (Semantic correctness of $\vec{\partial}$ (limited)). *For any term $x_1 : \mathbf{real}, \dots, x_n : \mathbf{real} \vdash t : \mathbf{real}$, the function $\llbracket \vec{\partial}(t) \rrbracket$ is the dual numbers representation (2) of $\llbracket t \rrbracket$. In detail: for any smooth functions $f_1 \dots f_n : \mathbb{R} \rightarrow \mathbb{R}$,*

$$(f_1, \nabla f_1, \dots, f_n, \nabla f_n; \llbracket \vec{\partial}(t) \rrbracket) = ((f_1 \dots f_n); \llbracket t \rrbracket, \nabla((f_1 \dots f_n); \llbracket t \rrbracket)).$$

(For instance, if $n = 2$, then $\llbracket \vec{\partial}(t) \rrbracket(x_1, 1, x_2, 0) = (\llbracket t \rrbracket(x_1, x_2), \frac{\partial \llbracket t \rrbracket(x, x_2)}{\partial x}(x_1)).$)

Proof. We prove this by logical relations. Although the following proof is elementary, we found it by using the categorical methods in § 5.

For each type τ , we define a binary relation S_τ between curves in $\llbracket \tau \rrbracket$ and curves in $\llbracket \vec{\mathcal{D}}(\tau) \rrbracket$, i.e. $S_\tau \subseteq \mathcal{P}_{\llbracket \tau \rrbracket}^{\mathbb{R}} \times \mathcal{P}_{\llbracket \vec{\mathcal{D}}(\tau) \rrbracket}^{\mathbb{R}}$, by induction on τ :

- $S_{\mathbf{real}} \stackrel{\text{def}}{=} \{(f, (f, \nabla f)) \mid f : \mathbb{R} \rightarrow \mathbb{R} \text{ smooth}\};$
- $S_{(\tau * \sigma)} \stackrel{\text{def}}{=} \{((f_1, g_1), (f_2, g_2)) \mid (f_1, f_2) \in S_\tau, (g_1, g_2) \in S_\sigma\};$
- $S_{\tau \rightarrow \sigma} \stackrel{\text{def}}{=} \{(f_1, f_2) \mid \forall (g_1, g_2) \in S_\tau. (x \mapsto f_1(x)(g_1(x)), x \mapsto f_2(x)(g_2(x))) \in S_\sigma\}.$

Then, we establish the following ‘fundamental lemma’:

If $x_1 : \tau_1, \dots, x_n : \tau_n \vdash t : \sigma$ and, for all $1 \leq i \leq n$, $y_1 \dots y_m : \mathbf{real} \vdash s_i : \tau_i$ is such that $((f_1, \dots, f_m); \llbracket s_i \rrbracket, (f_1, \nabla f_1), \dots, f_m, \nabla f_m); \llbracket \vec{\mathcal{D}}(s_i) \rrbracket \in S_{\tau_i}$ for all smooth $f_i : \mathbb{R} \rightarrow \mathbb{R}$, then $((f_1, \dots, f_m); \llbracket t[s_1/x_1, \dots, s_n/x_n] \rrbracket, (f_1, \nabla f_1), \dots, f_m, \nabla f_m); \llbracket \vec{\mathcal{D}}(t[s_1/x_1, \dots, s_n/x_n]) \rrbracket$ is in S_σ for all smooth $f_i : \mathbb{R} \rightarrow \mathbb{R}$.

This is proved routinely by induction on the typing derivation of t . The case for $*$ relies on the precise definition of $\vec{\mathcal{D}}(t * s)$, and similarly for $+$, ς .

We conclude the theorem from the fundamental lemma by considering the case where $\tau_i = \sigma = \mathbf{real}$, $m = n$ and $s_i = y_i$. \square

4 Extending the language: variant and inductive types

In this section, we show that the definition of forward AD and the semantics generalize if we extend the language of §2 with variants and inductive types. As an example of inductive types, we consider lists. This specific choice is only for expository purposes and the whole development works at the level of generality of arbitrary algebraic data types generated as initial algebras of (polynomial) type constructors formed by finite products and variants.

Similarly, our choice of operations is for expository purposes. More generally, assume given a family of operations $(\mathbf{Op}_n)_{n \in \mathbb{N}}$ indexed by their arity n . Further assume that each $\mathbf{op} \in \mathbf{Op}_n$ has type $\mathbf{real}^n \rightarrow \mathbf{real}$. We then ask for a certain closure of these operations under differentiation, that is we define

$$\vec{\mathcal{D}}(\mathbf{op}(t_1, \dots, t_n)) \stackrel{\text{def}}{=} \text{case } \vec{\mathcal{D}}(t_1) \text{ of } \langle x_1, x'_1 \rangle \rightarrow \dots \rightarrow \text{case } \vec{\mathcal{D}}(t_n) \text{ of } \langle x_n, x'_n \rangle \rightarrow \langle \mathbf{op}(x_1, \dots, x_n), \sum_{i=1}^n x'_i * \partial_i \mathbf{op}(x_1, \dots, x_n) \rangle$$

where $\partial_i \mathbf{op}(x_1, \dots, x_n)$ is some chosen term in the language, involving free variables from x_1, \dots, x_n , which we think of as implementing the partial derivative of \mathbf{op} with respect to its i -th argument. For constructing the semantics, every \mathbf{op} must be interpreted by some smooth function, and, to establish correctness, the semantics of $\partial_i \mathbf{op}(x_1, \dots, x_n)$ must be the semantic i -th partial derivative of the semantics of $\mathbf{op}(x_1, \dots, x_n)$.

Language. We additionally consider the following types and terms:

$$\begin{array}{lll} \tau, \sigma, \rho ::= & \text{types} & \mid \text{list}(\tau) \quad \text{list} \\ & & \mid \{ \ell_1 \tau_1 \mid \dots \mid \ell_n \tau_n \} \quad \text{variant} \end{array}$$

$t, s, r ::=$	terms
$\tau.\ell t$	variant constructor
$[] \mid t :: s$	empty list and cons
$\mathbf{case} t \mathbf{of} \{\ell_1 x_1 \rightarrow s_1 \mid \dots \mid \ell_n x_n \rightarrow s_n\}$	pattern matching: variants
$\mathbf{fold} (x_1, x_2).t \mathbf{over} s \mathbf{from} r$	list fold

We extend the type system according to:

$$\begin{array}{c}
\frac{\Gamma \vdash t : \tau_i \quad ((\ell_i \tau_i) \in \tau)}{\Gamma \vdash \tau.\ell_i t : \tau} \quad \frac{}{\Gamma \vdash [] : \mathbf{list}(\tau)} \quad \frac{\Gamma \vdash t : \tau \quad \Gamma \vdash s : \mathbf{list}(\tau)}{\Gamma \vdash t :: s : \mathbf{list}(\tau)} \\
\frac{\Gamma \vdash t : \{\ell_1 \tau_1 \mid \dots \mid \ell_n \tau_n\} \quad \text{for each } 1 \leq i \leq n: \Gamma, x_i : \tau_i \vdash s_i : \tau}{\Gamma \vdash \mathbf{case} t \mathbf{of} \{\ell_1 x_1 \rightarrow s_1 \mid \dots \mid \ell_n x_n \rightarrow s_n\} : \tau} \\
\frac{\Gamma \vdash s : \mathbf{list}(\tau) \quad \Gamma \vdash r : \sigma \quad \Gamma, x_1 : \tau, x_2 : \sigma \vdash t : \sigma}{\Gamma \vdash \mathbf{fold} (x_1, x_2).t \mathbf{over} s \mathbf{from} r : \sigma}
\end{array}$$

We can then extend $\vec{\mathcal{D}}$ to our new types and terms by

$$\begin{aligned}
\vec{\mathcal{D}}(\{\ell_1 \tau_1 \mid \dots \mid \ell_n \tau_n\}) &\stackrel{\text{def}}{=} \{\ell_1 \vec{\mathcal{D}}(\tau_1) \mid \dots \mid \ell_n \vec{\mathcal{D}}(\tau_n)\} & \vec{\mathcal{D}}(\mathbf{list}(\tau)) &\stackrel{\text{def}}{=} \mathbf{list}(\vec{\mathcal{D}}(\tau)) \\
\vec{\mathcal{D}}(\tau.\ell t) &\stackrel{\text{def}}{=} \vec{\mathcal{D}}(\tau).\ell \vec{\mathcal{D}}(t) & \vec{\mathcal{D}}([]) &\stackrel{\text{def}}{=} [] & \vec{\mathcal{D}}(t :: s) &\stackrel{\text{def}}{=} \vec{\mathcal{D}}(t) :: \vec{\mathcal{D}}(s) \\
\vec{\mathcal{D}}(\mathbf{case} t \mathbf{of} \{\ell_1 x_1 \rightarrow s_1 \mid \dots \mid \ell_n x_n \rightarrow s_n\}) &\stackrel{\text{def}}{=} \\
&\quad \mathbf{case} \vec{\mathcal{D}}(t) \mathbf{of} \{\ell_1 x_1 \rightarrow \vec{\mathcal{D}}(s_1) \mid \dots \mid \ell_n x_n \rightarrow \vec{\mathcal{D}}(s_n)\} \\
\vec{\mathcal{D}}(\mathbf{fold} (x_1, x_2).t \mathbf{over} s \mathbf{from} r) &\stackrel{\text{def}}{=} \mathbf{fold} (x_1, x_2).\vec{\mathcal{D}}(t) \mathbf{over} \vec{\mathcal{D}}(s) \mathbf{from} \vec{\mathcal{D}}(r)
\end{aligned}$$

To demonstrate the practical use of expressive type systems for differential programming, we consider the following two examples.

Example 6 (Lists of inputs for neural nets). Usually, we run a neural network on a large data set, the size of which might be determined at runtime. To evaluate a neural network on multiple inputs, in practice, one often sums the outcomes. This can be coded in our extended language as follows. Suppose that we have a network $f : (\mathbf{real}^n * P) \rightarrow \mathbf{real}$ that operates on single input vectors. We can construct one that operates on lists of inputs as follows:

$$g \stackrel{\text{def}}{=} \lambda \langle l, w \rangle. \mathbf{fold} (x_1, x_2). f \langle x_1, w \rangle + x_2 \mathbf{over} l \mathbf{from} \underline{0} : (\mathbf{list}(\mathbf{real}^n) * P) \rightarrow \mathbf{real}$$

Example 7 (Missing data). In practically every application of statistics and machine learning, we face the problem of *missing data*: for some observations, only partial information is available. In an expressive typed programming language like we consider, we can model missing data conveniently using the data type $\mathbf{maybe}(\tau) = \{\mathbf{Nothing}() \mid \mathbf{Just} \tau\}$. In the context of a neural network, one might use it as follows. First, define some helper functions

$$\begin{aligned}
\mathbf{fromMaybe} &\stackrel{\text{def}}{=} \lambda x. \lambda m. \mathbf{case} m \mathbf{of} \{\mathbf{Nothing} _ \rightarrow x \mid \mathbf{Just} x' \rightarrow x'\} \\
\mathbf{fromMaybe}^n &\stackrel{\text{def}}{=} \lambda \langle x_1, \dots, x_n \rangle. \lambda \langle m_1, \dots, m_n \rangle. \langle \mathbf{fromMaybe} x_1 m_1, \dots, \mathbf{fromMaybe} x_n m_n \rangle \\
&\quad : (\mathbf{maybe}(\mathbf{real}))^n \rightarrow \mathbf{real}^n \rightarrow \mathbf{real}^n \\
\mathbf{map} &\stackrel{\text{def}}{=} \lambda f. \lambda l. \mathbf{fold} (x_1, x_2). f x_1 :: x_2 \mathbf{over} l \mathbf{from} [] : (\tau \rightarrow \sigma) \rightarrow \mathbf{list}(\tau) \rightarrow \mathbf{list}(\sigma)
\end{aligned}$$

Given a neural network $f : (\mathbf{list}(\mathbf{real}^k) * P) \rightarrow \mathbf{real}$, we can build a new one that operates on a data set for which some covariates (features) are missing, by passing in default values to replace the missing covariates:

$$\lambda \langle l, \langle m, w \rangle \rangle . f \langle \text{map} (\text{fromMaybe}^k m) l, w \rangle \\ : (\mathbf{list}((\mathbf{maybe}(\mathbf{real}))^k) * (\mathbf{real}^k * P)) \rightarrow \mathbf{real}$$

Then, given a data set l with missing covariates, we can perform automatic differentiation on this network to optimize, simultaneously, the ordinary network parameters w and the default values for missing covariates m .

Semantics. In § 3 we gave a denotational semantics for the simple language in diffeological spaces. This extends to the language in this section, as follows. As before, each type τ is interpreted as a diffeological space, which is a set equipped with a family of plots:

- A variant type $\{\ell_1 \tau_1 \mid \dots \mid \ell_n \tau_n\}$ is inductively interpreted as the disjoint union $\llbracket \{\ell_1 \tau_1 \mid \dots \mid \ell_n \tau_n\} \rrbracket \stackrel{\text{def}}{=} \uplus_{i=1}^n \llbracket \tau_i \rrbracket$ with U -plots
$$\mathcal{P}_{\llbracket \{\ell_1 \tau_1 \mid \dots \mid \ell_n \tau_n\} \rrbracket}^U \stackrel{\text{def}}{=} \left\{ \left[U_j \xrightarrow{f_j} \llbracket \tau_j \rrbracket \rightarrow \uplus_{i=1}^n \llbracket \tau_i \rrbracket \right]_{j=1}^n \mid U = \uplus_{j=1}^n U_j, f_j \in \mathcal{P}_{\llbracket \tau_j \rrbracket}^{U_j} \right\}.$$
- A list type $\mathbf{list}(\tau)$ is interpreted as the set of lists, $\llbracket \mathbf{list}(\tau) \rrbracket \stackrel{\text{def}}{=} \uplus_{i=1}^\infty \llbracket \tau \rrbracket^i$ with U -plots
$$\mathcal{P}_{\llbracket \mathbf{list}(\tau) \rrbracket}^U \stackrel{\text{def}}{=} \left\{ \left[U_j \xrightarrow{f_j} \llbracket \tau \rrbracket^j \rightarrow \uplus_{i=1}^\infty \llbracket \tau \rrbracket^i \right]_{j=1}^\infty \mid U = \uplus_{j=1}^\infty U_j, f_j \in \mathcal{P}_{\llbracket \tau \rrbracket^j}^{U_j} \right\}.$$

The constructors and destructors for variants and lists are interpreted as in the usual set theoretic semantics. It is routine to show inductively that these interpretations are smooth. Thus every term $\Gamma \vdash t : \tau$ in the extended language is interpreted as a smooth function $\llbracket t \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket \tau \rrbracket$ between diffeological spaces.

(In this section we focused on a language with lists, but other inductive types are easily interpreted in the category of diffeological spaces in much the same way; the categorically minded reader may regard this as a consequence of **Diff** being a concrete Grothendieck quasitopos, e.g. [3].)

5 Categorical analysis of forward AD and its correctness

This section has three parts. First, we give a categorical account of the functoriality of AD (Ex. 8). Then we introduce our gluing construction, and relate it to the correctness of AD (dgm. (3)). Finally, we state and prove a correctness theorem for all first order types by considering a category of manifolds (Th. 2).

Syntactic categories. Our language induces a syntactic category as follows.

Definition 2. Let **Syn** be the category whose objects are types, and where a morphism $\tau \rightarrow \sigma$ is a term in context $x : \tau \vdash t : \sigma$ modulo the $\beta\eta$ -laws (Fig. 4). Composition is by substitution.

For simplicity, we do not impose arithmetic identities such as $x + y = y + x$ in **Syn**. As is standard, this category has the following universal property.

Lemma 2 (e.g. [27]). *For every bicartesian closed category \mathcal{C} with list objects, and every object $F(\mathbf{real}) \in \mathcal{C}$ and morphisms $F(\underline{c}) \in \mathcal{C}(1, F(\mathbf{real}))$, $F(+), F(*) \in \mathcal{C}(F(\mathbf{real}) \times F(\mathbf{real}), F(\mathbf{real}))$, $F(\varsigma) \in \mathbf{Syn}(F(\mathbf{real}), F(\mathbf{real}))$ in \mathcal{C} , there is a unique functor $F : \mathbf{Syn} \rightarrow \mathcal{C}$ respecting the interpretation and preserving the bicartesian closed structure as well as list objects.*

Proof (notes). The functor $F : \mathbf{Syn} \rightarrow \mathcal{C}$ is a canonical denotational semantics for the language, interpreting types as objects of \mathcal{C} and terms as morphisms. For instance, $F(\tau \rightarrow \sigma) \stackrel{\text{def}}{=} (F\tau \rightarrow F\sigma)$, the function space in the category \mathcal{C} , and $F(ts) \stackrel{\text{def}}{=} (Ft, Fs); \text{eval}$. When $\mathcal{C} = \mathbf{Diff}$, the denotational semantics of the language in diffeological spaces (§3,4) can be understood as the unique structure preserving functor $\llbracket - \rrbracket : \mathbf{Syn} \rightarrow \mathbf{Diff}$ satisfying $\llbracket \mathbf{real} \rrbracket = \mathbb{R}$, $\llbracket \varsigma \rrbracket = \varsigma$ and so on. \square

Example 8 (Canonical definition forward AD). The forward AD macro $\vec{\mathcal{D}}$ (§2,4) arises as a canonical cartesian closed functor on **Syn**. Consider the unique cartesian closed functor $F : \mathbf{Syn} \rightarrow \mathbf{Syn}$ such that $F(\mathbf{real}) = \mathbf{real} * \mathbf{real}$, $F(\underline{c}) = \vec{\mathcal{D}}(\underline{c})$, $F(\varsigma) = \vec{\mathcal{D}}(\varsigma(x))$, and

$$F(+) = z : F(\mathbf{real}) * F(\mathbf{real}) \vdash \text{case } z \text{ of } \langle x, y \rangle \rightarrow \vec{\mathcal{D}}(x + y) : F(\mathbf{real})$$

$$F(*) = z : F(\mathbf{real}) * F(\mathbf{real}) \vdash \text{case } z \text{ of } \langle x, y \rangle \rightarrow \vec{\mathcal{D}}(x * y) : F(\mathbf{real})$$

Then for any type τ , $F(\tau) = \vec{\mathcal{D}}(\tau)$, and for any term $x : \tau \vdash t : \sigma$, $F(t) = \vec{\mathcal{D}}(t)$ as morphisms $F(\tau) \rightarrow F(\sigma)$ in the syntactic category.

Categorical gluing and logical relations. Gluing is a method for building new categorical models which has been used for many purposes, including logical relations and realizability [24]. Our logical relations argument in the proof of Th. 1 can be understood in this setting. In this subsection, for the categorically minded, we explain this, and in doing so we quickly recover a correctness result for the more general language in § 4 and for arbitrary first order types.

We define a category \mathbf{Gl}_U whose objects are triples (X, X', S) where X and X' are diffeological spaces and $S \subseteq \mathcal{P}_X^U \times \mathcal{P}_{X'}^U$, is a relation between their U -plots. A morphism $(X, X', S) \rightarrow (Y, Y', T)$ is a pair of smooth functions

$\begin{aligned} & \text{case } \langle t_1, \dots, t_n \rangle \text{ of } \langle x_1, \dots, x_n \rangle \rightarrow s = s[t_1/x_1, \dots, t_n/x_n] \\ & s[t/y] \stackrel{\#x_1, \dots, x_n}{=} \text{case } t \text{ of } \langle x_1, \dots, x_n \rangle \rightarrow s[\langle x_1, \dots, x_n \rangle / y] \\ & \text{case } \ell_i \text{ of } \{ \ell_1 x_1 \rightarrow s_1 \mid \dots \mid \ell_n x_n \rightarrow s_n \} = s_i[t/x_i] \\ & s[t/y] \stackrel{\#x_1, \dots, x_n}{=} \text{case } t \text{ of } \{ \ell_1 x_1 \rightarrow s[\ell_1 x_1 / y] \mid \dots \mid \ell_n x_n \rightarrow s[\ell_n x_n / y] \} \\ & \text{fold } (x_1, x_2).t \text{ over } [] \text{ from } r = r \\ & \text{fold } (x_1, x_2).t \text{ over } s_1 :: s_2 \text{ from } r = t[s_1/x_1, \text{fold } (x_1, x_2).t \text{ over } s_2 \text{ from } r / x_2] \\ & u = s[\ell/y], r[s/x_2] = s[x_1::y/y] \Rightarrow s[t/y] \stackrel{\#x_1, x_2}{=} \text{fold } (x_1, x_2).r \text{ over } t \text{ from } u \end{aligned}$	$\begin{aligned} & (\lambda x.t) s = t[s/x] \\ & t \stackrel{\#x}{=} \lambda x.t x \end{aligned}$ <p style="font-size: small;">We write $\#x_1, \dots, x_n$ to indicate that the variables are free in the left hand side.</p>
---	---

Fig. 4. Standard $\beta\eta$ -laws (e.g. [27]) for products, functions, variants and lists.

$f: X \rightarrow Y$, $f': X' \rightarrow Y'$, such that if $(g, g') \in S$ then $(g; f, g'; f') \in T$. The idea is that this is a semantic domain where we can simultaneously interpret the language and its automatic derivatives.

Proposition 1. *The category \mathbf{Gl}_U is bicartesian closed, has list objects, and the projection functor $\text{proj}: \mathbf{Gl}_U \rightarrow \mathbf{Diff} \times \mathbf{Diff}$ preserves this structure.*

Proof (notes). The category \mathbf{Gl}_U is a full subcategory of the comma category $\text{id}_{\mathbf{Set}} \downarrow \mathbf{Diff}(U, -) \times \mathbf{Diff}(U, -)$. The result thus follows by the general theory of categorical gluing (e.g. [17, Lemma 15]). \square

We give a semantics $\llbracket - \rrbracket = (\llbracket - \rrbracket_0, \llbracket - \rrbracket_1, S_-)$ for the language in $\mathbf{Gl}_{\mathbb{R}}$, interpreting types τ as objects $(\llbracket \tau \rrbracket_0, \llbracket \tau \rrbracket_1, S_\tau)$, and terms as morphisms. We let $\llbracket \mathbf{real} \rrbracket_0 \stackrel{\text{def}}{=} \mathbb{R}$ and $\llbracket \mathbf{real} \rrbracket_1 \stackrel{\text{def}}{=} \mathbb{R}^2$, with the relation $S_{\mathbf{real}} \stackrel{\text{def}}{=} \{(f, (f, \nabla f)) \mid f: \mathbb{R} \rightarrow \mathbb{R} \text{ smooth}\}$. We interpret the constants \underline{c} as pairs $\llbracket \underline{c} \rrbracket_0 \stackrel{\text{def}}{=} \underline{c}$ and $\llbracket \underline{c} \rrbracket_1 \stackrel{\text{def}}{=} (\underline{c}, 0)$, and we interpret $+$, \times , ς in the standard way (meaning, like $\llbracket - \rrbracket$) in $\llbracket - \rrbracket_0$, but according to the derivatives in $\llbracket - \rrbracket_1$, for instance, $\llbracket (*) \rrbracket_1: \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \mathbb{R}^2$ is

$$\llbracket (*) \rrbracket_1((x, x'), (y, y')) \stackrel{\text{def}}{=} (xy, xy' + x'y).$$

At this point one checks that these interpretations are indeed morphisms in $\mathbf{Gl}_{\mathbb{R}}$. This amounts to checking that these interpretations are dual numbers representations in the sense of (2). The remaining constructions of the language are interpreted using the categorical structure of $\mathbf{Gl}_{\mathbb{R}}$, following Lem. 2.

Notice that the diagram below commutes. One can check this by hand or note that it follows from the initiality of \mathbf{Syn} (Lem. 2): all the functors preserve all the structure.

$$\begin{array}{ccc} \mathbf{Syn} & \xrightarrow{(\text{id}, \vec{\mathcal{D}}(-))} & \mathbf{Syn} \times \mathbf{Syn} \\ \llbracket - \rrbracket \downarrow & & \downarrow \llbracket - \rrbracket \times \llbracket - \rrbracket \\ \mathbf{Gl}_{\mathbb{R}} & \xrightarrow{\text{proj}} & \mathbf{Diff} \times \mathbf{Diff} \end{array} \quad (3)$$

We thus arrive at a restatement of the correctness theorem (Th. 1), which holds even for the extended language with variants and lists, because for any $x_1 \dots x_n: \mathbf{real} \vdash t: \mathbf{real}$, the interpretations $(\llbracket t \rrbracket, \llbracket \vec{\mathcal{D}}(t) \rrbracket)$ are in the image of the projection $\mathbf{Gl}_{\mathbb{R}} \rightarrow \mathbf{Diff} \times \mathbf{Diff}$, and hence $\llbracket \vec{\mathcal{D}}(t) \rrbracket$ is a dual numbers encoding of $\llbracket t \rrbracket$.

Correctness at all first order types, via manifolds. We now generalize Theorem 1 to hold at all first order types, not just the reals. To do this, we need to define the derivative of a smooth map between the interpretations of first order types. We do this by recalling the well known theory of manifolds and tangent bundles.

For our purposes, a smooth manifold M is a second-countable Hausdorff topological space together with a smooth atlas: an open cover \mathcal{U} together with homeomorphisms $(\phi_U: U \rightarrow \mathbb{R}^{n(U)})_{U \in \mathcal{U}}$ (called charts) such that $\phi_U^{-1}; \phi_V$ is smooth

on its domain of definition for all $U, V \in \mathcal{U}$. A function $f : M \rightarrow N$ between manifolds is smooth if $\phi_U^{-1}; f; \psi_V$ is smooth for all charts ϕ_U and ψ_V of M and N , respectively. Let us write **Man** for this category.

Our manifolds are slightly unusual because different charts in an atlas may have different finite dimension $n(U)$. Thus we consider manifolds with dimensions that are potentially unbounded, albeit locally finite. This does not affect the theory of differential geometry as far as we need it here.

Each open subset of \mathbb{R}^n can be regarded as a manifold. This lets us regard the category of manifolds **Man** as a full subcategory of the category of diffeological spaces. We consider a manifold $(X, \{\phi_U\}_U)$ as a diffeological space with the same carrier set X and where the plots \mathcal{P}_X^U are the smooth functions in **Man** (U, X) . A function $X \rightarrow Y$ is smooth in the sense of manifolds if and only if it is smooth in the sense of diffeological spaces [16]. For the categorically minded reader, this means that we have a full embedding of **Man** into **Diff**. Moreover, the natural interpretation of the first order fragment of our language in **Man** coincides with that in **Diff**. That is, the embedding of **Man** into **Diff** preserves finite products and countable coproducts (hence initial algebras of polynomial endofunctors).

Proposition 2. *Suppose that a type τ is first order, i.e. it is just built from reals, products, variants, and lists (or, again, arbitrary inductive types), and not function types. Then the diffeological space $\llbracket \tau \rrbracket$ is a manifold.*

Proof (notes). This is proved by induction on the structure of types. In fact, one may show that every such $\llbracket \tau \rrbracket$ is isomorphic to a manifold of the form $\biguplus_{i=1}^n \mathbb{R}^{d_i}$ where the bound n is either finite or ∞ , but this isomorphism is typically not an identity function. \square

The constraint to first order types is necessary because, e.g. the space $\llbracket \mathbf{real} \rightarrow \mathbf{real} \rrbracket$ is not a manifold, because of a Borsuk-Ulam argument (see [15], Appx. A).

We recall that the derivative of any morphism $f : M \rightarrow N$ of manifolds is given as follows. For each point x in a manifold M , define the *tangent space* $\mathcal{T}_x M$ to be the set $\{\gamma \in \mathbf{Man}(\mathbb{R}, M) \mid \gamma(0) = x\} / \sim$ of equivalence classes $[\gamma]$ of smooth curves γ in M based at x , where we identify $\gamma_1 \sim \gamma_2$ iff $\nabla(\gamma_1; f)(0) = \nabla(\gamma_2; f)(0)$ for all smooth $f : M \rightarrow \mathbb{R}$. The *tangent bundle* of M is the set $\mathcal{T}(M) \stackrel{\text{def}}{=} \biguplus_{x \in M} \mathcal{T}_x(M)$. The charts of M equip $\mathcal{T}(M)$ with a canonical manifold structure. Then for smooth $f : M \rightarrow N$, the derivative $\mathcal{T}(f) : \mathcal{T}(M) \rightarrow \mathcal{T}(N)$ is defined as $\mathcal{T}(f)(x, [\gamma]) \stackrel{\text{def}}{=} (f(x), [\gamma; f])$. All told, the derivative is a functor $\mathcal{T} : \mathbf{Man} \rightarrow \mathbf{Man}$.

As is standard, we can understand the tangent bundle of a composite space in terms of that of its parts.

Lemma 3. *There are canonical isomorphisms $\mathcal{T}(\biguplus_{i=1}^\infty M_i) \cong \biguplus_{i=1}^\infty \mathcal{T}(M_i)$ and $\mathcal{T}(M_1 \times \dots \times M_n) \cong \mathcal{T}(M_1) \times \dots \times \mathcal{T}(M_n)$.*

We define a canonical isomorphism $\phi_{\vec{\mathcal{T}}}^{\vec{\mathcal{T}}} : \llbracket \vec{\mathcal{T}}(\tau) \rrbracket \rightarrow \mathcal{T}(\llbracket \tau \rrbracket)$ for every type τ , by induction on the structure of types. We let $\phi_{\mathbf{real}}^{\vec{\mathcal{T}}} : \llbracket \vec{\mathcal{T}}(\mathbf{real}) \rrbracket \rightarrow \mathcal{T}(\llbracket \mathbf{real} \rrbracket)$ be

given by $\phi_{\mathbf{real}}^{\vec{\mathcal{D}}\tau}(x, x') \stackrel{\text{def}}{=} (x, [t \mapsto x + x't])$. For the other types, we use Lemma 3. We can now phrase correctness at all first order types.

Theorem 2 (Semantic correctness of $\vec{\mathcal{D}}$ (full)). *For any ground τ , any first order context Γ and any term $\Gamma \vdash t : \tau$, the syntactic translation $\vec{\mathcal{D}}$ coincides with the tangent bundle functor, modulo these canonical isomorphisms:*

$$\begin{array}{ccc} \llbracket \vec{\mathcal{D}}(\Gamma) \rrbracket & \xrightarrow{\llbracket \vec{\mathcal{D}}(t) \rrbracket} & \llbracket \vec{\mathcal{D}}(\tau) \rrbracket \\ \phi_{\Gamma}^{\vec{\mathcal{D}}\tau} \downarrow \cong & & \cong \downarrow \phi_{\tau}^{\vec{\mathcal{D}}\tau} \\ \mathcal{T}(\llbracket \Gamma \rrbracket) & \xrightarrow{\mathcal{T}(\llbracket t \rrbracket)} & \mathcal{T}(\llbracket \tau \rrbracket) \end{array}$$

Proof (notes). For any curve $\gamma \in \mathbf{Man}(\mathbb{R}, M)$, let $\bar{\gamma} \in \mathbf{Man}(\mathbb{R}, \mathcal{T}(M))$ be the tangent curve, given by $\bar{\gamma}(x) = (\gamma(x), [t \mapsto \gamma(x + t)])$. First, we note that a smooth map $h : \mathcal{T}(M) \rightarrow \mathcal{T}(N)$ is of the form $\mathcal{T}(g)$ for some $g : M \rightarrow N$ if for all smooth curves $\gamma : \mathbb{R} \rightarrow M$ we have $\bar{\gamma}; h = (\gamma; g) : \mathbb{R} \rightarrow \mathcal{T}(N)$. This generalizes (2). Second, for any first order type τ , $S_{[\tau]} = \{(f, \tilde{f}) \mid \tilde{f}; \phi_{\tau}^{\vec{\mathcal{D}}\tau} = \tilde{f}\}$. This is shown by induction on the structure of types. We conclude the theorem from diagram (3), by putting these two observations together. \square

6 A continuation-based AD algorithm

We now illustrate the flexibility of our framework by briefly describing an alternative syntactic translation $\vec{\mathcal{D}}_{\rho}$. This alternative translation uses aspects of continuation passing style, inspired by recent developments in reverse mode AD [34, 5]. In brief, $\vec{\mathcal{D}}_{\rho}$ works by $\vec{\mathcal{D}}_{\rho}(\mathbf{real}) = (\mathbf{real} * (\mathbf{real} \rightarrow \rho))$. Thus instead of using a pair of a number and its tangent, we use a pair of a number and a continuation. The answer type $\rho = \mathbf{real}^k$ needs to have the structure of a vector space, and the continuations that we consider will turn out to be linear maps. Because we work in continuation passing style, the chain rule is applied contravariantly. If the reader is familiar with reverse-mode AD algorithms, they may think of the dimension k as the number of memory cells used to store the result.

Computing the whole gradient of a term $x_1 : \mathbf{real}, \dots, x_k : \mathbf{real} \vdash t : \mathbf{real}$ at once is then achieved by running $\vec{\mathcal{D}}_k(t)$ on a k -tuple of basis vectors for \mathbf{real}^k .

We define the continuation-based AD macro $\vec{\mathcal{D}}_k$ on types and terms as the unique structure preserving functor $\mathbf{Syn} \rightarrow \mathbf{Syn}$ with $\vec{\mathcal{D}}_k(\mathbf{real}) = (\mathbf{real} * (\mathbf{real} \rightarrow \mathbf{real}^k))$ and

$$\begin{aligned} \vec{\mathcal{D}}_k(c) &\stackrel{\text{def}}{=} \langle c, \lambda z. \langle 0, \dots, 0 \rangle \rangle \\ \vec{\mathcal{D}}_k(t + s) &\stackrel{\text{def}}{=} \mathbf{case} \vec{\mathcal{D}}_k(t) \mathbf{of} \langle x, x' \rangle \rightarrow \mathbf{case} \vec{\mathcal{D}}_k(s) \mathbf{of} \langle y, y' \rangle \rightarrow \langle x + y, \lambda z. x' z + y' z \rangle \\ \vec{\mathcal{D}}_k(t * s) &\stackrel{\text{def}}{=} \mathbf{case} \vec{\mathcal{D}}_k(t) \mathbf{of} \langle x, x' \rangle \rightarrow \mathbf{case} \vec{\mathcal{D}}_k(s) \mathbf{of} \langle y, y' \rangle \rightarrow \\ &\quad \langle x * y, \lambda z. x' (y * z) + y' (x * z) \rangle \\ \vec{\mathcal{D}}_k(\varsigma(t)) &\stackrel{\text{def}}{=} \mathbf{case} \vec{\mathcal{D}}_k(t) \mathbf{of} \langle x, x' \rangle \rightarrow \mathbf{let} y = \varsigma(x) \mathbf{in} \langle y, \lambda z. x' (y * (1 - y) * z) \rangle. \end{aligned}$$

Here, we use sugar $x : \mathbf{real}^k, y : \mathbf{real}^k \vdash x + y \stackrel{\text{def}}{=} \mathbf{case} x \mathbf{of} \langle x_1, \dots, x_k \rangle \rightarrow$

case y **of** $\langle y_1, \dots, y_k \rangle \rightarrow \langle x_1 + y_1, \dots, x_k + y_k \rangle$. (We could easily expand this definition by making $\overleftarrow{\mathcal{D}}_k$ preserve all other term and type formers, as we did for $\overrightarrow{\mathcal{D}}$.) Note that the corresponding scheme for an arbitrary n -ary operation **op** would be (c.f. the scheme for forward AD in §4)

$$\overleftarrow{\mathcal{D}}_k(\text{op}(t_1, \dots, t_n)) \stackrel{\text{def}}{=} \text{case } \overleftarrow{\mathcal{D}}_k(t_1) \text{ of } \langle x_1, x'_1 \rangle \rightarrow \dots \rightarrow \text{case } \overleftarrow{\mathcal{D}}_k(t_n) \text{ of } \langle x_n, x'_n \rangle \rightarrow \langle \text{op}(x_1, \dots, x_n), \lambda z. \sum_{i=1}^n x'_i (\partial_i \text{op}(x_1, \dots, x_n) * z) \rangle.$$

The idea is that $\overleftarrow{\mathcal{D}}_k(t)$ is a higher order function that simultaneously computes t (the forward pass) and defines as a continuation the reverse pass which computes the gradient. In order to actually run the algorithm, we need two auxiliary definitions

$$\text{lamR}_{\text{real}}^k \stackrel{\text{def}}{=} \lambda z. \text{case } z \text{ of } \langle x, x' \rangle \rightarrow \text{case } x' \text{ of } \langle x'_1, \dots, x'_k \rangle \rightarrow \langle x, \lambda y. \langle x'_1 * y, \dots, x'_k * y \rangle \rangle : \overrightarrow{\mathcal{D}}_k(\text{real}) \rightarrow \overleftarrow{\mathcal{D}}_k(\text{real})$$

$$\text{evR}_{\text{real}}^k \stackrel{\text{def}}{=} \lambda z. \text{case } z \text{ of } \langle x, x' \rangle \rightarrow \langle x, x' \underline{1} \rangle : \overleftarrow{\mathcal{D}}_k(\text{real}) \rightarrow \overrightarrow{\mathcal{D}}_k(\text{real}).$$

Here, $\overrightarrow{\mathcal{D}}_k$ is a macro on types (and terms) with exactly the same inductive definition as $\overrightarrow{\mathcal{D}}$ except for the base case $\overrightarrow{\mathcal{D}}_k(\text{real}) = (\text{real} * \text{real}^k)$. By noting that both $\overrightarrow{\mathcal{D}}_k$ and $\overleftarrow{\mathcal{D}}_k$ preserve all type formers, we can extend these definitions to all first order types τ : $z : \overrightarrow{\mathcal{D}}_k(\tau) \vdash \text{lamR}_\tau^k(z) : \overleftarrow{\mathcal{D}}_k(\tau)$, $z : \overleftarrow{\mathcal{D}}_k(\tau) \vdash \text{evR}_\tau^k(z) : \overrightarrow{\mathcal{D}}_k(\tau)$. We can think of $\text{lamR}_\tau^k(z)$ as encoding k tangent vectors $z : \overrightarrow{\mathcal{D}}_k(\tau)$ as a closure, so it is suitable for running $\overleftarrow{\mathcal{D}}_k(t)$ on, and $\text{evR}_\tau^k(z)$ as actually evaluating the reverse pass defined by $z : \overleftarrow{\mathcal{D}}_k(\tau)$ and returning the result as k tangent vectors. The idea is that given some $x : \tau \vdash t : \sigma$ between first order types τ, σ , we run our continuation-based AD by running $\text{evR}_\sigma^k(\overleftarrow{\mathcal{D}}_k(t) [\text{lamR}_\tau^k(z)/x])$.

The correctness proof closely follows that for forward AD. In particular, one defines a binary logical relation $(\text{real})^{r,k} = (\mathbb{R}, \mathbb{R} \times (\mathbb{R}^k)^{\mathbb{R}}, S_{\text{real}}^{r,k})$, where $S_{\text{real}}^{r,k} = \left\{ (f, x \mapsto (f(x), y \mapsto (\partial_1 f(x) * y, \dots, \partial_k f(x) * y))) \mid f \in \mathcal{P}_{\mathbb{R}}^{\mathbb{R}^k} \right\}$, on the plots $\mathcal{P}_{\mathbb{R}}^{\mathbb{R}^k} \times \mathcal{P}_{\mathbb{R} \times ((\mathbb{R}^k)^{\mathbb{R}})}^{\mathbb{R}^k}$ and verifies that $\llbracket \underline{c} \rrbracket \times \llbracket \overleftarrow{\mathcal{D}}_k(\underline{c}) \rrbracket$, $\llbracket x + y \rrbracket \times \llbracket \overleftarrow{\mathcal{D}}_k(x + y) \rrbracket$, $\llbracket x * y \rrbracket \times \llbracket \overleftarrow{\mathcal{D}}_k(x * y) \rrbracket$ and $\llbracket \varsigma(x) \rrbracket \times \llbracket \overleftarrow{\mathcal{D}}_k(\varsigma(x)) \rrbracket$ respect this logical relation. It follows that this relation extends to a functor $(-)^{r,k} : \mathbf{Syn} \rightarrow \mathbf{GI}_{\mathbb{R}^k}$ such that $\text{id} \times \overleftarrow{\mathcal{D}}_k$ factors over $(-)^{r,k}$, implying the correctness of the continuation-based AD by the following lemma.

Lemma 4. *For all first order types τ (i.e. types not involving function types), we have that $\llbracket \text{evR}_\tau^k(\text{lamR}_\tau^k(t)) \rrbracket = \llbracket t \rrbracket$.*

Proof (notes). This follows by an induction on the structure of τ . The idea is that lamR_τ^k embeds reals into function spaces as linear maps, which is undone by evR_τ^k by evaluating the linear maps at $\underline{1}$. \square

To phrase correctness, in this setting, however, we need a few definitions. Keeping in mind the canonical projection $\mathcal{T}(M) \rightarrow M$, we define $\mathcal{T}^k(M)$ as the k -fold categorical pullback (fibre product) $\mathcal{T}(M) \times_M \dots \times_M \mathcal{T}(M)$. To be explicit, $\mathcal{T}_x^k M$ consists of k -tuples of tangent vectors at the base point x . Again, \mathcal{T}^k extends to a functor $\mathbf{Man} \rightarrow \mathbf{Man}$ by defining $\mathcal{T}^k(f)(x, (v_1, \dots, v_k)) \stackrel{\text{def}}{=} (f(x), (\mathcal{T}_x(f)(v_1), \dots, \mathcal{T}_x(f)(v_k)))$. As \mathcal{T}^k preserves countable coproducts and

finite products (like \mathcal{T}), it follows that the isomorphisms $\phi_{\tau}^{\vec{\mathcal{D}}}$ generalize to canonical isomorphisms $\phi_{\tau,k}^{\vec{\mathcal{D}}} : \llbracket \vec{\mathcal{D}}_k(\tau) \rrbracket \rightarrow \mathcal{T}^k(\llbracket \tau \rrbracket)$ for first order types τ . This leads to the following correctness statement for continuation-based AD.

Theorem 3 (Semantic correctness of $\vec{\mathcal{D}}_k$). *For any ground τ , any first order context Γ and any term $\Gamma \vdash t : \tau$, syntactic translation $t \mapsto \text{evR}_{\tau}^k(\vec{\mathcal{D}}_k(t)[\text{lamR}_{\tau}^k(z)/\dots])$ coincides with the tangent bundle functor, modulo these canonical isomorphisms:*

$$\begin{array}{ccc} \llbracket \vec{\mathcal{D}}_k(\Gamma) \rrbracket & \xrightarrow{\llbracket \text{lamR}_{\Gamma}^k; \vec{\mathcal{D}}_k(t); \text{evR}_{\tau}^k \rrbracket} & \llbracket \vec{\mathcal{D}}_k(\tau) \rrbracket \\ \phi_{\Gamma,k}^{\vec{\mathcal{D}}} \downarrow \cong & & \cong \downarrow \phi_{\tau,k}^{\vec{\mathcal{D}}} \\ \mathcal{T}^k(\llbracket \Gamma \rrbracket) & \xrightarrow{\mathcal{T}^k(\llbracket t \rrbracket)} & \mathcal{T}^k(\llbracket \tau \rrbracket) \end{array}$$

For example, when $\tau = \mathbf{real}$ and $\Gamma = x, y : \mathbf{real}$, we can run our continuation-based AD to compute the gradient of a program $x, y : \mathbf{real} \vdash t : \mathbf{real}$ at values $x = V, y = W$ by evaluating

$$\text{evR}_{\mathbf{real}}^2(\vec{\mathcal{D}}_2(t)[(\text{lamR}_{x:\mathbf{real}}^2 v)/_x, (\text{lamR}_{y:\mathbf{real}}^2 w)/_y])(\langle V, \langle \underline{1}, \underline{0} \rangle \rangle /_v, \langle W, \langle \underline{0}, \underline{1} \rangle \rangle /_w).$$

Indeed,

$$\begin{aligned} & \llbracket \text{evR}_{\mathbf{real}}^2(\vec{\mathcal{D}}_2(t)[(\text{lamR}_{x:\mathbf{real}}^2 v)/_x, (\text{lamR}_{y:\mathbf{real}}^2 w)/_y])(\langle V, \langle \underline{1}, \underline{0} \rangle \rangle /_v, \langle W, \langle \underline{0}, \underline{1} \rangle \rangle /_w) \rrbracket = \\ & (\llbracket t \rrbracket(\llbracket V \rrbracket, \llbracket W \rrbracket), \partial_1 \llbracket t \rrbracket(\llbracket V \rrbracket, \llbracket W \rrbracket), \partial_2 \llbracket t \rrbracket(\llbracket V \rrbracket, \llbracket W \rrbracket)). \end{aligned}$$

7 Discussion and future work

Summary. We have shown that diffeological spaces provide a denotational semantics for a higher order language with variants and inductive types (§3.4). We have used this to show correctness of a simple AD translation (Thm. 1, Thm. 2). But the method is not tied to this specific translation, as we illustrated in Section 6.

The structure of our elementary correctness argument for Theorem 1 is a typical logical relations proof. As explained in Section 5, this can equivalently be understood as a denotational semantics in a new kind of space obtained by categorical gluing.

Overall, then, there are two logical relations at play. One is in diffeological spaces, which ensures that all definable functions are smooth. The other is in the correctness proof (equivalently in the categorical gluing), which explicitly tracks the derivative of each function, and tracks the syntactic AD even at higher types.

Connection to the state of the art in AD implementation. As is common in denotational semantics research, we have here focused on an idealized language and simple translations to illustrate the main aspects of the method. There are a number of points where our approach is simplistic compared to the advanced current practice, as we now explain.

Representation of vectors. In our examples we have treated n -vectors as tuples of length n . This style of programming does not scale to large n . A better solution would be to use array types, following [31]. Our categorical semantics and correctness proofs straightforwardly extend to cover them, in a similar way to our treatment of lists.

Efficient forward-mode AD. For AD to be useful, it must be fast. The syntactic translation $\vec{\mathcal{D}}$ that we use is the basis of an efficient AD library [31]. However, numerous optimizations are needed, ranging from algebraic manipulations, to partial evaluations, to the use of an optimizing C compiler. A topic for future work would be to validate some of these manipulations using our semantics. The resulting implementation is performant in experiments [31].

Efficient reverse-mode AD. Our sketch of continuation-based AD is primarily intended to emphasise that our denotational approach is not tied to any specific translation $\vec{\mathcal{D}}$. Nonetheless, it is worth noting that this algorithm shares similarities with advanced reverse-mode implementations: (1) it calculates derivatives in a (contravariant) “reverse pass” in which derivatives of operations are evaluated in the reverse order compared to their order in calculating the function value; (2) it can be used to calculate the full gradient of a function $\mathbb{R}^n \rightarrow \mathbb{R}$ in a single reverse pass (while n passes of fwd AD would be necessary). However, it lacks important optimizations and the continuation scales with the size of the input n where it should scale with the size of the output. This adds an important overhead, as pointed out in [26]. Speed being the main attraction of reverse-mode AD, its implementations tend to rely on mutable state, control operators and/or staging [26, 6, 34, 5], which we have not considered here.

Other language features. The idealized languages that we considered so far do not touch on several useful language constructs. For example: the use of functions that are partial (such as division) or partly-smooth (such as ReLU); phenomena such as iteration, recursion; and probabilities. There are suggestions that the denotational approach using diffeological spaces can be adapted to these features using standard categorical methods. We leave this for future work.

Acknowledgements. We have benefited from discussing this work with many people, including B. Pearlmutter, O. Kammar, C. Mak, L. Ong, G. Plotkin, A. Shaikhha, J. Sigal, and others. Our work is supported by the Royal Society and by a Facebook Research Award. In the course of this work, MV has also been employed at Oxford (EPSRC Project EP/M023974/1) and at Columbia in the Stan development team. This project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 895827.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: A system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). pp. 265–283 (2016)
2. Abadi, M., Plotkin, G.D.: A simple differentiable programming language. In: Proc. POPL 2020. ACM (2020)
3. Baez, J., Hoffnung, A.: Convenient categories of smooth spaces. Transactions of the American Mathematical Society **363**(11), 5789–5825 (2011)
4. Barthe, G., Crubillé, R., Lago, U.D., Gavazzo, F.: On the versatility of open logical relations: Continuity, automatic differentiation, and a containment theorem. In: Proc. ESOP 2020. Springer (2020), to appear
5. Brunel, A., Mazza, D., Pagani, M.: Backpropagation in the simply typed lambda-calculus with linear negation. In: Proc. POPL 2020 (2020)
6. Carpenter, B., Hoffman, M.D., Brubaker, M., Lee, D., Li, P., Betancourt, M.: The Stan math library: Reverse-mode automatic differentiation in C++. arXiv preprint arXiv:1509.07164 (2015)
7. Christensen, J.D., Wu, E.: Tangent spaces and tangent bundles for diffeological spaces. arXiv preprint arXiv:1411.5425 (2014)
8. Cockett, J.R.B., Cruttwell, G.S.H., Gallagher, J., Lemay, J.S.P., MacAdam, B., Plotkin, G.D., Pronk, D.: Reverse derivative categories. In: Proc. CSL 2020 (2020)
9. Cruttwell, G., Gallagher, J., MacAdam, B.: Towards formalizing and extending differential programming using tangent categories. In: Proc. ACT 2019 (2019)
10. Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research **12**(Jul), 2121–2159 (2011)
11. Ehrhard, T., Regnier, L.: The differential lambda-calculus. Theoretical Computer Science **309**(1-3), 1–41 (2003)
12. Elliott, C.: The simple essence of automatic differentiation. Proceedings of the ACM on Programming Languages **2**(ICFP), 70 (2018)
13. Fong, B., Spivak, D., Tuyéras, R.: Backprop as functor: A compositional perspective on supervised learning. In: 2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). pp. 1–13. IEEE (2019)
14. Hoffman, M.D., Gelman, A.: The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. Journal of Machine Learning Research **15**(1), 1593–1623 (2014)
15. Huot, M., Staton, S., Vákár, M.: Correctness of automatic differentiation via diffeologies and categorical gluing. Full version (2020), arxiv:2001.02209
16. Iglesias-Zemmour, P.: Diffeology. American Mathematical Soc. (2013)
17. Johnstone, P.T., Lack, S., Sobocinski, P.: Quasitoposes, quasiadhesive categories and Artin glueing. In: Proc. CALCO 2007 (2007)
18. Kiefer, J., Wolfowitz, J., et al.: Stochastic estimation of the maximum of a regression function. The Annals of Mathematical Statistics **23**(3), 462–466 (1952)
19. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
20. Kucukelbir, A., Tran, D., Ranganath, R., Gelman, A., Blei, D.M.: Automatic differentiation variational inference. The Journal of Machine Learning Research **18**(1), 430–474 (2017)

21. Liu, D.C., Nocedal, J.: On the limited memory BFGS method for large scale optimization. *Mathematical programming* **45**(1-3), 503–528 (1989)
22. Mak, C., Ong, L.: A differential-form pullback programming language for higher-order reverse-mode automatic differentiation (2020), [arxiv:2002.08241](https://arxiv.org/abs/2002.08241)
23. Manzyuk, O.: A simply typed λ -calculus of forward automatic differentiation. In: *Proc. MFPS 2012* (2012)
24. Mitchell, J.C., Scedrov, A.: Notes on scoping and relators. In: *International Workshop on Computer Science Logic*. pp. 352–378. Springer (1992)
25. Neal, R.M., et al.: MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo* **2**(11), 2 (2011)
26. Pearlmutter, B.A., Siskind, J.M.: Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **30**(2), 7 (2008)
27. Pitts, A.M.: Categorical logic. Tech. rep., University of Cambridge, Computer Laboratory (1995)
28. Plotkin, G.D.: Some principles of differential programming languages (2018), invited talk, POPL 2018
29. Qian, N.: On the momentum term in gradient descent learning algorithms. *Neural networks* **12**(1), 145–151 (1999)
30. Robbins, H., Monro, S.: A stochastic approximation method. *The annals of mathematical statistics* pp. 400–407 (1951)
31. Shaikhha, A., Fitzgibbon, A., Vytiniotis, D., Peyton Jones, S.: Efficient differentiable programming in a functional array-processing language. *Proceedings of the ACM on Programming Languages* **3**(ICFP), 97 (2019)
32. Souriau, J.M.: Groupes différentiels. In: *Differential geometrical methods in mathematical physics*, pp. 91–128. Springer (1980)
33. Stacey, A.: Comparative smootheology. *Theory Appl. Categ.* **25**(4), 64–117 (2011)
34. Wang, F., Wu, X., Essertel, G., Decker, J., Rompf, T.: Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proceedings of the ACM on Programming Languages* **3**(ICFP) (2019)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

