Operational Calculus on Programming Spaces

Žiga Sajovic, Martin Vuk

July 17, 2018

Abstract

In this work we present a theoretical model for differentiable programming. We present an algebraic language that enables both implementations and analysis of differentiable programs (of arbitrary order of differentition).

To this purpose, we develop an *abstract computational model of automatically differentiable programs* of arbitrary order. In the model, programs are elements of op. cit. *programming spaces* and are viewed as maps from a finite-dimensional vector space to itself op. cit. *virtual memory space*. Virtual memory space is also an algebra of programs, *an algebraic data structure* one can calculate with.

We define the *operator of differentiation* (∂) on programming spaces and, using its powers, implement the *general shift operator* and the *operator of program composition*. We provide the formula for the expansion of a differentiable program into an infinite tensor series in terms of the powers of ∂. We express the operator of program composition in terms of the generalized shift operator and ∂, which implements a differntiable composition in the language. We prove that our language enables differentiation of the derivatives of programs by the use of the *order reduction map*.

The provided higher order constructs are sufficient for the language to be a complete model of differentiable programming.

1 Introduction

According to John Backus Von Neumann languages do not have useful properties for reasoning about programs. Axiomatic and denotational semantics are precise tools for describing and understanding conventional programs, but they only talk about them and cannot alter their ungainly properties [1]. This issue has partially been addressed by algebraic data types employed by functional programming, where a mapping has been shown between grammars and semirings [2]. Yet due to the lack of inverses (hence the semiring structure) we remain limited in the algebraic manipulations we are allowed to employ [3].

As computer programs are the dominant tool for modern problem solving, the need for examining the analytic properties of programs led to the develop-

ment of various tools for dealing with derivatives of computer programs (automatic differentiation). Yet the developed techniques are only efficient ways of calculating derivatives, and do not construct any algebraic structure over differentiable programs. As such, there is still a need for a framework that enables algebraic investigations of differentiable programs.

In recent times, names like *Differentiable Programming* and *Software* 2.0 have attached themselves to Deep learning, as it has shown itself to be more than a collection of machine learning algorithms; it is emerging as a new programming paradigm. But because the field is still in its youth, most of the advances come as a result of empirical investigations. Yet, as it is founded on rigorous mathematical objects, it offers an opportunity to be formalized as a language. And as it is rooted in tensor algebra, it holds the ability to address the outlined issues, because *unlike von Neumann languages, the language of ordinary algebra is suitable both for stating its laws and for transforming an equation into its solution, all within the language* [1]. Furthermore, due of its innate relationship with calculus, a language encompassing it would serve as a formal gateway of analysis into programming.

TODO: Finish the introduction, notes in the .tex

2 Computer Programs as Maps on a Vector Space

We will model computer programs as maps on a vector space. If we only focus on the real valued variables (of type `float` or `double`), the state of the memory can be seen as a high dimensional vector[1]. A set of all the possible states of the program's memory, can be modeled by a finite dimensional real vector space $\mathcal{V} \equiv \mathbb{R}^n$. We will call $\mathcal{V}$ the *memory space of the program.* The effect of a computer program on its memory space $\mathcal{V}$, can be described by a map

$$P : \mathcal{V} \to \mathcal{V}. \tag{1}$$

A programming space is a space of maps $\mathcal{V} \to \mathcal{V}$ that can be implemented as a program in specific programming language.

Definition 2.1 (Euclidean machine)**.** *The tuple* $(\mathcal{V}, \mathcal{F})$ *is an Euclidean machine, where*

- $\mathcal{V}$ *is a finite dimensional vector space over a complete field* K*, serving as memory*[2]
- $\mathcal{F} < \mathcal{V}^{\mathcal{V}}$ *is a subspace of the space of maps* $\mathcal{V} \to \mathcal{V}$*, called* programming space*, serving as actions on the memory.*

At first glance, the *Euclidean machine* seems like a description of functional programming, with its compositions inherited from $\mathcal{F}$. An intended impression,

[1] We assume the variables of interest to be of type `float` for simplicity. Theoretically any field can be used instead of $\mathbb{R}$.

[2] In most applications the field K will be $\mathbb{R}$

as we wish for the *Euclidean machine* to build on its elegance. But note that in the coming section an additional restriction is imposed on $\mathcal{F}$; that of its elements being differentiable.

3 Differentiable Maps and Programs

To define differentiable programs, let us first recall some definitions from multivariate calculus.

Definition 3.1 (Derivative)**.** *Let V, U be Banach spaces. A map $P : V \to U$ is differentiable at a point $\mathbf{x} \in V$, if there exists a linear bounded operator $TP_{\mathbf{x}} : V \to U$ such that*

$$\lim_{\mathbf{h}\to 0} \frac{\|P(\mathbf{x}+\mathbf{h}) - P(\mathbf{x}) - TP_{\mathbf{x}}(\mathbf{h})\|}{\|\mathbf{h}\|} = 0. \tag{2}$$

The map $TP_{\mathbf{x}}$ is called the Fréchet derivative *of the map P at the point $\mathbf{x}$.*

For maps $\mathbb{R}^n \to \mathbb{R}^m$ Fréchet derivative can be expressed by multiplication of vector $\mathbf{h}$ by the Jacobi matrix $\mathbf{J}_{P,\mathbf{x}}$ of partial derivatives of the components of the map P

$$T_{\mathbf{x}}P(\mathbf{h}) = \mathbf{J}_{P,\mathbf{x}} \cdot \mathbf{h}.$$

We assume for the remainder of this section that the map $P : V \to U$ is differentiable for all $\mathbf{x} \in V$. The derivative defines a map from V to linear bounded maps from V to U. We further assume U and V are finite dimensional. Then the space of linear maps from V to U is isomorphic to the tensor product $U \otimes V^*$, where the isomorphism is given by the tensor contraction, sending a simple tensor $\mathbf{u} \otimes f \in U \otimes V^*$ to a linear map

$$\mathbf{u} \otimes f : \mathbf{x} \mapsto f(\mathbf{x}) \cdot \mathbf{u}. \tag{3}$$

The derivative defines a map

$$\partial P \quad : \quad V \to U \otimes V^* \tag{4}$$
$$\partial P \quad : \quad \mathbf{x} \mapsto T_{\mathbf{x}}P. \tag{5}$$

One can consider the differentiability of the derivative itself ∂P by looking at it as a map (4). This leads to the definition of the higher derivatives.

Definition 3.2 (higher derivatives)**.** *Let $P : V \to U$ be a map from vector space V to vector space U. The derivative $\partial^k P$ of order k of the map P is the map*

$$\partial^k P \quad : \quad V \to U \otimes (V^*)^{\otimes k} \tag{6}$$
$$\partial^k P \quad : \quad \mathbf{x} \mapsto T_{\mathbf{x}}\left(\partial^{k-1} P\right) \tag{7}$$

Remark 3.1. *For the sake of clarity, we assumed in the definition above, that the map P as well as all its derivatives are differentiable at all points $\mathbf{x}$. If this is not the case, definitions above can be done locally, which would introduce mostly technical difficulties.*

Let $\mathbf{e}_1, \ldots, \mathbf{e}_n$ be a basis of U and $x_1, \ldots x_m$ the basis of V^*. Denote by $P_i = x_i \circ P$ the $i-th$ component of the map P according to the basis $\{\mathbf{e}_i\}$ of U. Then $\partial^k P$ can be defined in terms of directional (partial) derivatives by the formula

$$\partial^k P = \sum_{\forall_{i,\alpha}} \frac{\partial^k P_i}{\partial x_{\alpha_1} \ldots \partial x_{\alpha_k}} \mathbf{e}_i \otimes dx_{\alpha_1} \otimes \ldots \otimes dx_{\alpha_k}. \tag{8}$$

3.1 Differentiable Programs

We want to be able to represent the derivatives of a computer program in an *Euclidean machine* again as a program in the same *Euclidean machine*. We define three subspaces of the virtual memory space $\mathcal{V}$, that describe how different parts of the memory influence the final result of the program.

Denote by $\mathbf{e}_1, \ldots \mathbf{e}_n$ a standard basis of the memory space $\mathcal{V}$ and by $x_1, \ldots x_n$ the dual basis of $\mathcal{V}^*$. The functions x_i are coordinate functions on $\mathcal{V}$ and correspond to individual locations(variables) in the program memory.

Definition 3.3. *For each program P in the programming space $\mathcal{F} < \mathcal{V}^{\mathcal{V}}$, we define the* input *or* parameter space $I_P < \mathcal{V}$ *and the* output space $O_P < \mathcal{V}$ *to be the minimal vector sub-spaces spanned by the standard basis vectors, such that the map P_e, defined by the following commutative diagram*

$$\begin{array}{ccc} \mathcal{V} & \xrightarrow{P} & \mathcal{V} \\ {\scriptstyle \vec{i} \mapsto \vec{i} + \vec{f}} \uparrow & & \downarrow {\scriptstyle \mathrm{pr}_{O_P}} \\ I_P & \xrightarrow{P_e} & O_P \end{array} \tag{9}$$

does not depend of the choice of the element $\vec{f} \in F_P = (I_P + O_P)^\perp$.

The space $F_P = (I_P + O_P)^\perp$ is called free space *of the program P.*

The variables x_i corresponding to standard basis vectors spanning the parameter, output and free space are called *paramters* or *input variables*, *output variables* and *free variables* correspondingly. Free variables are those that are left intact by the program and have no influence on the final result other than their value itself. The output of the program depends only on the values of the input variables and consists of variables that have changed during the program. Input parameters and output values might overlap.

The map P_e is called the *effective map* of the program P and describes the actual effect of the program P on the memory, ignoring the free memory.

The derivative of the effective map is of interest, when we speak about differentiability of computer programs.

Definition 3.4 (Automatically differentiable programs)**.** *A program $P : \mathcal{V} \to \mathcal{V}$ is* automatically differentiable *if there exist an embedding of the space $O_P \otimes I_P^*$ into the free space F_P, and a program $(1 + \partial P) : \mathcal{V} \to \mathcal{V}$, such that its effective map is the map*

$$P_e \oplus \partial P_e : I_P \to O_P \oplus (O_P \otimes I^*). \tag{10}$$

A program $P : \mathcal{V} \to \mathcal{V}$ is automatically differentiable of order k *if there exist a program $\tau_k P : \mathcal{V} \to \mathcal{V}$, such that its effective map is the map*

$$P_e \oplus \partial P_e \oplus \ldots \partial^k P_e : I_P \to O_P \oplus (O_P \otimes I^*) \oplus \ldots \left(O_P \otimes \left(I_p^*\right)^{k\otimes}\right). \tag{11}$$

If a program $P : \mathcal{V} \to \mathcal{V}$ is automatically differentiable then it is also differentiable as a map $\mathcal{V} \to \mathcal{V}$. However only the derivative of program's effective map can be implemented as a program, since the memory space is limited to $\mathcal{V}$. To be able to differentiate a program to the k-th order, we have to calculate and save all the derivatives of the orders k and less.

4 Differentiable Programming Spaces

The memory space of a program is rarely treated as more than a storage. But to endow the *Euclidean machine* with extra structure, this is precisely what to focus on. Loosely speaking, functional programming is described by monoids, and as such a (multi)linear algebra description of the memory space seems the appropriate step to take in attaining the wanted structure.

4.1 Memory space

Motivated by the Definition 3.4, we define the *memory space* for differentiable programs as a sequence of vector spaces with the recursive formula

$$\mathcal{V}_0 = \mathcal{V} \tag{12}$$
$$\mathcal{V}_k = \mathcal{V}_{k-1} + (\mathcal{V}_{k-1} \otimes \mathcal{V}^*). \tag{13}$$

Note that the sum is not direct, since some of the subspaces of $\mathcal{V}_{k-1}$ and $\mathcal{V}_{k-1} \otimes \mathcal{V}^*$ are naturally isomorphic and will be identified[3].

The space that satisfies the recursive formula (13) is

$$\mathcal{V}_k = \mathcal{V} \otimes \left(K \oplus \mathcal{V}^* \oplus (\mathcal{V}^* \otimes \mathcal{V}^*) \oplus \ldots (\mathcal{V}^*)^{\otimes k}\right) = \mathcal{V} \otimes T_k(\mathcal{V}^*), \tag{14}$$

where $T_k(\mathcal{V}^*)$ is a subspace of *tensor algebra* $T(\mathcal{V}^*)$, consisting of linear combinations of tensors of rank less or equal k. This construction enables us to define all the derivatives as maps with the same domain and codomain $\mathcal{V} \to \mathcal{V} \otimes T(\mathcal{V}^*)$.

As such, the memory space is a mapping from-and-to itself,

$$\mathcal{V}_n : \mathcal{V} \to \mathcal{V}, \tag{15}$$

which is, for an arbitrary $\mathbf{W} \in \mathcal{V}_n$, defined as,

$$\mathbf{W}(\mathbf{v}) = \mathbf{w}_0 + \mathbf{w}_1 \cdot \mathbf{v} + \cdots + \mathbf{w}_n \cdot (\mathbf{v})^{\otimes n}, \tag{16}$$

[3]The spaces $\mathcal{V} \otimes (\mathcal{V}^*)^{\otimes(j+1)}$ and $\mathcal{V} \otimes (\mathcal{V}^*)^{\otimes j} \otimes \mathcal{V}^*$ are naturally isomorphic and will be identified in the sum.

the sum of multiple contractions (where $\mathbf{w}_i \in \mathcal{V}_i$). The expression (16) will be rigorously defined in section 5.1. With such a construction, the expansions and contractions of the memory space (reminiscent to the breathing of the stack) would hold meaning parallel to storing values; which is what motives the next definition.

Definition 4.1 (Virtual memory space)**.** *Let $(\mathcal{V}, \mathcal{F})$ be an Euclidean machine and let*

$$\mathcal{V}_\infty = \mathcal{V} \otimes T(\mathcal{V}^*) = \mathcal{V} \oplus (\mathcal{V} \otimes \mathcal{V}^*) \oplus \dots, \tag{17}$$

where $T(\mathcal{V}^*)$ is the tensor algebra of the dual space $\mathcal{V}^*$. We call $\mathcal{V}_\infty$ the virtual memory space *of a* Euclidean machine *$(\mathcal{V}, \mathcal{F})$.*

The term virtual memory is used as it is only possible to embed certain subspaces of $\mathcal{V}_\infty$ into memory space $\mathcal{V}$, making it similar to virtual memory as a memory management technique.

We can extend each program $P : \mathcal{V} \to \mathcal{V}$ to the map on universal memory space $\mathcal{V}_\infty$ by setting the first component in the direct sum (17) to P, and all other components to zero. Similarly derivatives $\partial^k P$ can be also seen as maps from $\mathcal{V}$ to $\mathcal{V}_\infty$ by setting k-th component in the direct sum (17) to $\partial^k P$ and all others to zero.

4.2 Differentiable Programming Spaces

Let us define the following function spaces:

$$\mathcal{F}_n = \{f : \mathcal{V} \to \mathcal{V} \otimes T_n(\mathcal{V}^*)\} \tag{18}$$

All of these function spaces can be seen as sub spaces of $\mathcal{F}_\infty = \{f : \mathcal{V} \to \mathcal{V} \otimes T(\mathcal{V}^*)\}$, since $\mathcal{V}$ is naturally embedded into $\mathcal{V} \otimes T(\mathcal{V}^*)$. The Fréchet derivative defines an operator on the space of smooth maps in $\mathcal{F}_\infty$[4]. We denote this operator ∂. The image of any map $P : \mathcal{V} \to \mathcal{V}$ by operator ∂ is its first derivative, while the higher order derivatives are just powers of operator ∂ applied to P. Thus ∂^k is a mapping between function spaces (18)

$$\partial^k : \mathcal{F}^n \to \mathcal{F}^{n+k}. \tag{19}$$

Definition 4.2 (Differentiable programming space)**.** *A* differentiable programming space *$\mathcal{P}_0$ is any subspace of $\mathcal{F}_0$ such that*

$$\partial \mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(\mathcal{V}^*) \tag{20}$$

The space $\mathcal{P}_n < \mathcal{F}_n$ spanned by $\{\partial^k \mathcal{P}_0; \quad 0 \le k \le n\}$ over K, is called a differentiable programming space of order n. When all elements of $\mathcal{P}_0$ are analytic, we call $\mathcal{P}_0$ an analytic programming space*.*

[4]The operator ∂ may be defined partially for other maps as well, but we will handle this case later.

The definition of higher order differentiable programming spaces is justified by the following theorem.

Theorem 4.1 (Infinite differentiability)**.** *Any differentiable programming space $\mathcal{P}_0$ is an infinitely differentiable programming space, meaning that*

$$\partial^k \mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(\mathcal{V}^*) \tag{21}$$

for any $k \in \mathbb{N}$.

Proof. By induction on order k. For $k = 1$ the claim holds by definition. Assume $\forall_{P \in \mathcal{P}_0}, \partial^n \mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(\mathcal{V}^*)$. Denote by $P^i_{\alpha,k}$ the component of the k-th derivative for a multiindex α denoting the component of $T(\mathcal{V}^*)$ and an index i denoting the component of $\mathcal{V}$.

$$\partial^{n+1} P^i_{\alpha,k} = \partial(\partial^n P^i_\alpha)_k \wedge (\partial^n P^i_\alpha) \in \mathcal{P}_0 \implies \partial(\partial^n P^i_\alpha)_k \in \mathcal{P}_0 \otimes T(\mathcal{V}^*) \tag{22}$$

$$\implies$$

$$\partial^{n+1} \mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(\mathcal{V}^*)$$

Thus by induction, the claim holds for all $k \in \mathbb{N}$. □

Corollary 4.1.1. *A differentiable programming space of order n, $\mathcal{P}_n : \mathcal{V} \to \mathcal{V} \otimes T(\mathcal{V}^*)$, can be embedded into the tensor product of the function space $\mathcal{P}_0$ and the space $T_n(\mathcal{V}^*)$ of multi-tensors of order less than equal n:*

$$\mathcal{P}_n < \mathcal{P}_0 \otimes T_n(\mathcal{V}^*). \tag{23}$$

By taking the limit as $n \to \infty$, we consider

$$\mathcal{P}_\infty < \mathcal{P}_0 \otimes \mathcal{T}(\mathcal{V}^*), \tag{24}$$

where $\mathcal{T}(\mathcal{V}^*) = \prod_{k=0}^{\infty} (\mathcal{V}^*)^{\otimes k}$ is the *tensor series algebra*, the algebra of the infinite formal tensor series.[5]

4.3 Virtual Tensor Machine

We propose an abstract computational model, that is capable of constructing differentiable programming spaces. Such a model provides a framework for analytic study of programs by algebraic means.

Following from Theorem 4.1, the tuple $(\mathcal{V}, \mathcal{P}_0)$ – together with the structure of the tensor algebra $T(\mathcal{V}^*)$ – is sufficient for constructing differentiable programming spaces $\mathcal{P}_\infty$, using linear combinations of elements of $\mathcal{P}_0 \otimes T(\mathcal{V}^*)$. This motivates the following definition.

Definition 4.3 (Virtual tensor machine)**.** *The tuple $M = \langle \mathcal{V}, \mathcal{P}_0 \rangle$ is an analytic, infinitely differentiable virtual machine, where*

[5] The tensor series algebra is a completion of the tensor algebra $T(\mathcal{V}^*)$ in suitable topology.

- $\mathcal{V}$ *is a finite dimensional vector space*
- $\mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*)$ *is the virtual memory space*
- $\mathcal{P}_0$ *is an analytic programming space over* $\mathcal{V}$.

When composing contractions (16) of the memory with activation functions $\phi \in \mathcal{P}$, we note that fully connected *tensor networks*,

$$\mathcal{N}(v) = \phi_k \circ W_k \circ \cdots \circ \phi_0 \circ W_0(v), \tag{25}$$

are basic programs in a virtual tensor machine (the vanilla fully connected neural network is captured by the restriction $\forall_i(W_i \in \mathcal{V}_1)$). The formulation (25) is trivially generalized to convolutional models, but is omitted here for brevity.

5 Operational Calculus on Programming Spaces

By Corollary 4.1.1 we may represent calculation of derivatives of the map $P : \mathcal{V} \to \mathcal{V}$, with only one mapping τ. We define the operator τ_n as a direct sum of operators

$$\tau_n = 1 + \partial + \partial^2 + \ldots + \partial^n \tag{26}$$

The image $\tau_k P(\mathbf{x})$ is a multi-tensor of order k, which is a direct sum of the map's value and all derivatives of order $n \leq k$, all evaluated at the point $\mathbf{x}$:

$$\tau_k P(\mathbf{x}) = P(\mathbf{x}) + \partial_{\mathbf{x}} P(\mathbf{x}) + \partial^2_{\mathbf{x}} P(\mathbf{x}) + \ldots + \partial^k_{\mathbf{x}} P(\mathbf{x}). \tag{27}$$

The operator τ_n satisfies the recursive relation:

$$\tau_{k+1} = 1 + \partial \tau_k, \tag{28}$$

that can be used to recursively construct programming spaces of arbitrary order.

Proposition 5.1. *Only explicit knowledge of* $\tau_1 : \mathcal{P}_0 \to \mathcal{P}_1$ *is required for the construction of* $\mathcal{P}_n$ *from* $\mathcal{P}_1$.

Proof. The construction is achieved following the argument (22) of the proof of Theorem 4.1, allowing simple implementation, as dictated by (28). □

Remark 5.1. *Maps* $\mathcal{V} \otimes T(\mathcal{V}^*) \to \mathcal{V} \otimes T(\mathcal{V}^*)$ *are constructible using tensor algebra operations and compositions of programs in* $\mathcal{P}_n$.

Definition 5.1 (Algebra product)**.** *For any bilinear map*

$$\cdot : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$$

we can define a bilinear product $\cdot$ *on* $\mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*)$ *by the following rule on the simple tensors:*

$$(\mathbf{v} \otimes f_1 \otimes \ldots f_k) \cdot (\mathbf{u} \otimes g_1 \otimes \ldots g_l) = (\mathbf{v} \cdot \mathbf{u}) \otimes f_1 \otimes \ldots f_k \otimes g_1 \otimes \ldots g_l \tag{29}$$

extending linearly on the whole space $\mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*)$

Theorem 5.1 (Programming algebra)**.** *For any bilinear map* $\cdot : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$ *an infinitely-differentiable programming space* $\mathcal{P}_\infty$ *is a function algebra, with the product defined by* (29)*.*

5.1 Tensor Series Expansion

In th space spanned by the set $\mathcal{D}^n = \{\partial^k; \quad 0 \leq k \leq n\}$ over a field K, such an operator can be defined as

$$e^{h\partial} = \sum_{n=0}^{\infty} \frac{(h\partial)^n}{n!}$$

In coordinates, the operator $e^{h\partial}$ can be written as a series over all multi-indices α

$$e^{h\partial} = \sum_{n=0}^{\infty} \frac{h^n}{n!} \sum_{\forall_{i,\alpha}} \frac{\partial^n}{\partial x_{\alpha_1} \dots \partial x_{\alpha_n}} \mathbf{e}_i \otimes dx_{\alpha_1} \otimes \dots \otimes dx_{\alpha_n}. \tag{30}$$

The operator $e^{h\partial}$ is a mapping between function spaces (18)

$$e^{h\partial} : \mathcal{P} \to \mathcal{P}_\infty.$$

It also defines a map

$$e^{h\partial} : \mathcal{P} \times \mathcal{V} \to \mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*), \tag{31}$$

by taking the image of the map $e^{h\partial}(P)$ at a certain point $\mathbf{v} \in \mathcal{V}$. We may construct a map from the space of programs, to the space of polynomials using (31). Note that the space of multivariate polynomials $\mathcal{V} \to K$ is isomorphic to symmetric algebra $S(\mathcal{V}^*)$, which is in turn a quotient of tensor algebra $T(\mathcal{V}^*)$. To any element of $\mathcal{V} \otimes T(\mathcal{V}^*)$ one can attach corresponding element of $\mathcal{V} \otimes S(\mathcal{V}^*)$ namely a polynomial map $\mathcal{V} \to \mathcal{V}$. Thus, similarly to (24), we consider the completion of the symmetric algebra $S(\mathcal{V}^*)$ as the *formal power series* $\mathcal{S}(\mathcal{V}^*)$, which is in turn isomorphic to a quotient of *tensor series algebra* $\mathcal{T}(\mathcal{V}^*)$. This leads to

$$e^{h\partial} : \mathcal{P} \times \mathcal{V} \to \mathcal{V} \otimes \mathcal{S}(\mathcal{V}^*) \tag{32}$$

For any element $\mathbf{v}_0 \in \mathcal{V}$, the expression $e^{h\partial}(\cdot, \mathbf{v}_0)$ is a map $\mathcal{P} \to \mathcal{V} \otimes \mathcal{S}(\mathcal{V}^*)$, mapping a program to a formal power series.

We can express the correspondence between multi-tensors in $\mathcal{V} \otimes T(\mathcal{V}^*)$ and polynomial maps $\mathcal{V} \to \mathcal{V}$ given by multiple contractions for all possible indices. For a simple tensor $\mathbf{u} \otimes f_1 \otimes \dots \otimes f_n \in \mathcal{V} \otimes (\mathcal{V}^*)^{\otimes n}$ the contraction by $\mathbf{v} \in \mathcal{V}$ is given by applying co-vector f_n to $\mathbf{v}$ [6]

$$\mathbf{u} \otimes f_1 \otimes \dots \otimes f_n \cdot \mathbf{v} = f_n(\mathbf{v})\mathbf{u} \otimes f_1 \otimes \dots f_{n-1}. \tag{33}$$

By taking contraction multiple times, we can attach a monomial map to a simple tensor by

$$\mathbf{u} \otimes f_1 \otimes \dots \otimes f_n \cdot (\mathbf{v})^{\otimes n} = f_n(\mathbf{v}) f_{n-1}(\mathbf{v}) \cdots f_1(\mathbf{v})\mathbf{u}, \tag{34}$$

[6] For order two tensors from $\mathcal{V} \otimes \mathcal{V}^*$ the contraction corresponds to matrix vector multiplication.

Both contractions (33) and (34) are extended by linearity to spaces $\mathcal{V} \otimes (\mathcal{V}^*)^{\otimes n}$ and further to $\mathcal{V} \otimes T(\mathcal{V}^*)$.[7] For a multi-tensor $\mathbf{W} = \mathbf{w}_0 + \mathbf{w}_1 + \ldots + \mathbf{w}_n \in \mathcal{V} \otimes T_n(\mathcal{V}^*)$, where $\mathbf{w}_k \in \mathcal{V} \otimes (\mathcal{V}^*)^{\otimes k}$, applying the contraction by a vector $\mathbf{v} \in \mathcal{V}$ multiple times yields a polynomial map

$$\mathbf{W}(\mathbf{v}) = \mathbf{w}_0 + \mathbf{w}_1 \cdot \mathbf{v} + \ldots + \mathbf{w}_n \cdot (\mathbf{v})^{\otimes n}. \tag{35}$$

Theorem 5.2. *For a program $P \in \mathcal{P}$ the expansion into an infinite tensor series at the point $\mathbf{v}_0 \in \mathcal{V}$ is expressed by multiple contractions*

$$\begin{aligned} P(\mathbf{v}_0 + h\mathbf{v}) = \Big((e^{h\partial}P)(\mathbf{v}_0)\Big)(\mathbf{v}) &= \sum_{n=0}^{\infty} \frac{h^n}{n!} \partial^n P(\mathbf{v}_0) \cdot (\mathbf{v}^{\otimes n}) \\ &= \sum_{n=0}^{\infty} \frac{h^n}{n!} \sum_{\forall_{i,\alpha}} \frac{\partial^n P_i}{\partial x_{\alpha_1} \ldots \partial x_{\alpha_n}} \mathbf{e}_i \cdot dx_{\alpha_1}(\mathbf{v}) \cdot \ldots \cdot dx_{\alpha_n}(\mathbf{v}). \end{aligned} \tag{36}$$

Proof. We will show that $\frac{d^n}{dh^n}(\text{LHS})|_{h=0} = \frac{d^n}{dh^n}(\text{RHS})|_{h=0}$. Then LHS and RHS as functions of h have coinciding Taylor series and are therefore equal.

$\Longrightarrow$

$$\frac{d^n}{dh^n} P(\mathbf{v}_0 + h\mathbf{v})\Big|_{h=0} = \partial^n P(\mathbf{v}_0)(\mathbf{v})$$

$\Longleftarrow$

$$\frac{d^n}{dh^n} \left((e^{h\partial})(P)(\mathbf{v}_0)\right)(\mathbf{v})\Big|_{h=0} = \left((\partial^n e^{h\partial})(P)(\mathbf{v}_0)\right)(\mathbf{v})\big|_{h=0}$$

$$\wedge$$

$$\partial^n e^{h\partial}\big|_{h=0} = \sum_{i=0}^{\infty} \frac{h^i \partial^{i+n}}{i!}\Big|_{h=0} = \partial^n$$

$$\Longrightarrow$$

$$(\partial^n(P)(\mathbf{v}_0)) \cdot (\mathbf{v}^{\otimes n})$$

□

Remark 5.2. *Theorem 5.2 can be generalized to convolutions using the Volterra series [4].*

It follows trivially from the above theorem that the operator $e^{h\partial}$ is an automorphism of the programming algebra $\mathcal{P}_\infty$,

$$e^{h\partial}(p_1 \cdot p_2) = e^{h\partial}(p_1) \cdot e^{h\partial}(p_2) \tag{37}$$

where $\cdot$ stands for any bilinear map.

[7] Note that the simple order one tensor $\mathbf{u} \in \mathcal{V}$ can not be contracted by the vector $\mathbf{v}$. To be consistent we define $\mathbf{u} \cdot \mathbf{v} = \mathbf{u}$ and attach a constant map $\mathbf{v} \mapsto \mathbf{u}$ to order zero tensor $\mathbf{u}$. The extension of (34) to $\mathcal{V} \otimes T(\mathcal{V}^*)$ can be seen as a generalization of the affine map, where the zero order tensors account for translation.

Remark 5.3 (Generalized shift operator)**.** *The operator* $e^{h\partial} : \mathcal{P} \times \mathcal{V} \to \mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*)$ *evaluated at* $h = 1$ *is a broad generalization of the shift operator [5].*

For a specific $\mathbf{v}_0 \in \mathcal{V}$, the generalized shift operator is denoted by

$$e^{\partial}|_{\mathbf{v}_0} : \mathcal{P} \to \mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*)$$

When the choice of $\mathbf{v}_0 \in \mathcal{V}$ is arbitrary, we omit it from expressions for brevity.

5.2 Operator of Program Composition

In this section we implement the operator of program composition within the constructed algebraic language. Such a *composer* can than be used to implement the analog of the *U combinator* (which facilitates recursion) and other constructs. Furthermore, due to the differentiable nature of the employed objects, such a *composer* generalizes both forward (e.g. [6]) and reverse (e.g. [7]) mode of automatic differentiation of arbitrary order, unified under a single operator. We will than demonstrate how to perform calculations on the operator level, before they are applied to a particular programming space, which serves as an added level of abstraction.

Theorem 5.3 (Program composition)**.** *Composition of maps* $\mathcal{P}$ *is expressed as*

$$e^{h\partial}(f \circ g) = \exp(\partial_f e^{h\partial_g})(g, f) \tag{38}$$

where $\exp(\partial_f e^{h\partial_g}) : \mathcal{P} \times \mathcal{P} \to \mathcal{P}_\infty$ *is an operator on pairs of maps* (g, f)*, where* ∂_g *is differentiation operator applied to the first component* g*, and* ∂_f *to the second component* f*.*

Proof. We will show that $\frac{d^n}{dh^n}(\text{LHS})|_{h=0} = \frac{d^n}{dh^n}(\text{RHS})|_{h=0}$. Then LHS and RHS as functions of h have coinciding Taylor series and are therefore equal.

$\Longrightarrow$

$$\lim_{\|h\|\to 0}(\frac{d}{dh})^n e^{\partial}(f \circ g) = \lim_{\|h\|\to 0} \partial^n e^{h\partial}(f \circ g)$$
$$\Longrightarrow$$
$$\partial^n(f \circ g) \tag{39}$$

$\Longleftarrow$

$$\exp(\partial_f e^{h\partial_g}) = \exp\left(\partial_f \sum_{i=0}^{\infty} \frac{(h\partial_g)^i}{i!}\right) = \prod_{i=1}^{\infty} e^{\partial_f \frac{(h\partial_g)^i}{i!}}\left(e^{\partial_f}\right)$$
$$\Longrightarrow$$
$$\exp(\partial_f e^{h\partial_g})(g, f) = \sum_{\forall_n} h^n \sum_{\lambda(n)} \prod_{k\cdot l\in\lambda} \Big(\frac{\partial_f \partial_g^l(g)}{l!}\Big)^k \frac{1}{k!}\Big(\Big(e^{\partial_f}\Big)f\Big)$$

where $\lambda(n)$ stands for the partitions of n. Thus

$$\lim_{\|h\|\to 0}(\frac{d}{dh})^n \exp(\partial_f e^{h\partial_g}) = \sum_{\lambda(n)} n! \prod_{k\cdot l\in\lambda} \Big(\frac{\partial_f \partial_g^l(g)}{l!}\Big)^k \frac{1}{k!}\Big(\Big(e^{\partial_f}\Big)f\Big) \tag{40}$$

taking into consideration the fact that $e^{\partial_f}(f)$ evaluated at a point $\mathbf{v} \in \mathcal{V}$ is the same as evaluating f at $\mathbf{v}$, the expression (40) equals (39) by Faà di Bruno's formula.

$$\lim_{\|h\|\to 0}(\frac{d}{dh})^n \exp(\partial_f e^{h\partial_g}) = \sum_{\lambda(n)} n! \prod_{k\cdot l\in\lambda} \Big(\frac{\partial_f \partial_g^l(g(v))}{l!}\Big)^k \frac{1}{k!}\Big(f(g(\mathbf{v}))\Big) \tag{41}$$

□

The Theorem 5.3 enables an invariant implementation of the operator of program composition (i.e. the *composer*) in $\mathcal{P}_n$, expressed as a tensor series through (38) and (40).

By fixing the second map g in

$$\exp(\partial_f e^{h\partial_g}) : \mathcal{P} \times \mathcal{P} \to \mathcal{P}_\infty, \tag{42}$$

the operator

$$\exp(\partial_f e^{h\partial_g})(\cdot, g) = g^* \left(e^{h\partial}\right) \tag{43}$$

is the pullback through g of the generalized shift operator $e^{h\partial}$. While by fixing the first map f in (42), the operator

$$\exp(\partial_f e^{h\partial_g})(f, \cdot) = f_* \left(e^{h\partial}\right) \tag{44}$$

is the push-forward through f of the generalized shift operator $e^{h\partial}$. This also generalizes the *U combinator* to its forward and backward modes, by restring the *composers* (42) domain to a single function (i.e. f and g are the same mapping).

Remark 5.4 (Unified AD)**.** *Because of* (9) *and* (10) *every program can be seen as* $P = P_n \circ \dots P_1$*. Thus applying the operators* $\exp(\partial_f e^{h\partial_g})(\cdot, P_i)$ *from* $i = 1$ *to* $i = n$ *and projecting onto the space spanned by* $\{1, \partial\}$ *is equivalent to forward mode automatic differentiation, while applying the operators* $\exp(\partial_f e^{h\partial_g})(P_{n-i+1}, \cdot)$ *in reverse order (and projecting) is equivalent to reverse mode automatic differentiation.*

Corollary 5.3.1. *The operator* $e^{h\partial}$ *commutes with composition over* $\mathcal{P}$

$$e^{h\partial}(p_2 \circ p_1) = e^{h\partial}(p_2) \circ e^{h\partial}(p_1)$$

Proof. Follows from (32) and Theorem 5.3. □

Such calculations can be made easier, by completing them on the level of operators, thus avoiding the need to manipulate tensor series. This serves as a level of abstraction.

The derivative $\frac{d}{dh}$ of (43) is

$$\frac{d}{dh} \exp(\partial_f e^{h\partial_g})(g) = \partial_f(\partial_g g) e^{h\partial_g} \exp(\partial_f e^{h\partial_g})(g) \tag{45}$$

We note an important distinction to the operator $e^{h\partial_g}$, the derivative of which is

$$\frac{d}{dh}e^{h\partial_g} = \partial_g e^{h\partial_g} \tag{46}$$

We may now compute derivatives (of arbitrary order) of the *composer* itself.

5.3 Example of an Operator Level Computation

For illustrative purposes we compute the second derivative of the *composer* (38)

$$\left(\frac{d}{dh}\right)^2 \exp\left(\partial_f e^{h\partial_g}\right)(g) = \frac{d}{dh}\left(\partial_f(\partial_g g)e^{h\partial_g}\exp\left(\partial_f e^{h\partial_g}\right)(g)\right)$$

which is by equations (45) and (46), using algebra and correct applications equal to

$$\left(\partial_f(\partial_g^2 g)\right)e^{h\partial_g}\exp(\partial_f e^{h\partial_g})(g) + (\partial_f^2(\partial_g g)^2)e^{2h\partial_g}\exp(\partial_f e^{h\partial_g})(g) \tag{47}$$

The operator is always shifted to the evaluating point (31) $\mathbf{v} \in \mathcal{V}$, thus, only the behavior in the limit as $h \to 0$ is of importance. Taking this limit in the expression (47) we obtain the operator

$$\left(\partial_f(\partial_g^2 g) + \partial_f^2(\partial_g g)^2\right)\exp(\partial_f) : \mathcal{P} \to \partial^2\mathcal{P}(g)$$

Thus, without imposing any additional rules, we computed the operator of the second derivative of composition with g, directly on the level of operators. The result of course matches the equation (40) for $n = 2$.

As it is evident from the example, calculations using operators are far simpler, than direct manipulations of tensor series. This enables a simpler implementation that functions over arbitrary programming spaces. In the space that is spanned by $\{\partial^n\mathcal{P}_0\}$ over K, derivatives of compositions may be expressed solely through the operators, using only the product rule (37) and the derivative of the general shift operator (46). Thus, explicit knowledge of rules for differentiating compositions is unnecessary, as it is contained in the structure of the operator $exp(\partial_f e^{h\partial_g})$ itself, which is differentiated using standard rules, as shown by this example.

Similarly higher derivatives of the *composer* can be computed on the operator level

$$\partial^n(f \circ g) = \left(\frac{d}{dh}\right)^n \exp\left(\partial_f e^{h\partial_g}\right)(g, f)\Big|_{h=0}. \tag{48}$$

5.4 Automatically differentiable derivatives

The ability to use k-th derivative of a program $P_1 \in \mathcal{P}$ as part of a differentiable program $P_2 \in \mathcal{P}$ appears to be useful in many fields (e.g. [8]). For that to be sensible, we must be able to treat the (k-th) derivative itself as a differentiable program $P'^k \in \mathcal{P}$. This is what motivates the following theorem.

Theorem 5.4 (Order reduction)**.** *There exists a reduction of order map* $\phi : \mathcal{P}_n \to \mathcal{P}_{n-1}$*, such that the following diagram commutes*

$$\begin{array}{ccc} \mathcal{P}_n & \xrightarrow{\phi} & \mathcal{P}_{n-1} \\ \downarrow{\scriptstyle \partial} & & \downarrow{\scriptstyle \partial} \\ \mathcal{P}_{n+1} & \xrightarrow{\phi} & \mathcal{P}_n \end{array} \tag{49}$$

satisfying

$$\forall_{P_1 \in \mathcal{P}_0} \exists_{P_2 \in \mathcal{P}_0} \left(\phi^k \circ e_n^{\partial}(P_1) = e_{n-k}^{\partial}(P_2) \right)$$

for each $n \geq 1$*, where* e_n^{∂} *is the projection of the operator* e^{∂} *onto the set* $\{\partial^n\}$*.*

Corollary 5.4.1 (Differentiable derivative)**.** *By Theorem 5.4,* n*-differentiable* k*-th derivatives of a program* $P \in \mathcal{P}_0$ *can be extracted by*

$${}^{n}P^{k\prime} = \phi^k \circ e_{n+k}^{\partial}(P) \in \mathcal{P}_n$$

Thus, by corollary 5.4.1, the ability of writing differentiable programs that act on derivatives of other programs in well defined within the language. This is a crucial feature, as stressed by other authors [9, 10]. Please note that in order to use k-th derivative of P_2 in an n-differentiable program P_1, then P_2 must have been $(k+n)$-differentiable before ϕ^k was applied to it.

5.5 Control Structures

Until now, we restricted ourselves to operations, that change the memories' content. Along side assignment statements, we know control statements (ex. statements `if`, `for`, `while`, ...). Control statements don't directly influence values of variables, but change the execution tree of the program. This is why we interpret control structures as a piecewise-definition of a map. Each control structure divides the space of parameters into different domains, in which the execution of the program is always the same. The entire program divides the space of all possible parameters to a finite set of domains $\{\Omega_i; \quad i = 1, \dots k\}$, where the programs' execution is always the same. As such, a program may in general be piecewise-defined. For $\mathbf{v} \in \mathcal{V}$

$$P(\mathbf{v}) = \begin{cases} P_{n_1 1} \circ P_{(n_1-1)1} \circ \cdots P_{11}(\mathbf{v}); & \mathbf{v} \in \Omega_1 \\ P_{n_2 2} \circ P_{(n_2-1)2} \circ \cdots P_{12}(\mathbf{v}); & \mathbf{v} \in \Omega_2 \\ \vdots & \vdots \\ P_{n_k k} \circ P_{(n_k-1)k} \circ \cdots P_{1k}(\mathbf{v}); & \mathbf{v} \in \Omega_k \end{cases} \tag{50}$$

The operator e^{∂} (at some point) of a program P, is of course dependent on initial parameters $\mathbf{v}$, and can also be expressed piecewise inside domains Ω_i

$$e^{\partial}P(\mathbf{v}) = \begin{cases} e^{\partial}P_{n_1 1} \circ e^{\partial}P_{(n_1-1)1} \circ \cdots e^{\partial}P_{11}(\mathbf{v}); & \mathbf{v} \in \text{int}(\Omega_1) \\ e^{\partial}P_{n_2 2} \circ e^{\partial}P_{(n_2-1)2} \circ \cdots e^{\partial}P_{12}(\mathbf{v}); & \mathbf{v} \in \text{int}(\Omega_2) \\ \vdots & \vdots \\ e^{\partial}P_{n_k k} \circ e^{\partial}P_{(n_k-1)k} \circ \cdots e^{\partial}P_{1k}(\mathbf{v}); & \mathbf{v} \in \text{int}(\Omega_k) \end{cases} \tag{51}$$

Theorem 5.5. *Each program $P \in \mathcal{P}$ containing control structures is infinitely-differentiable on the domain $\Omega = \bigcup_{\forall_i} \text{int}(\Omega_i)$.*

Proof. Interior of each domain Ω_i is open. As the entire domain $\Omega = \bigcup_{\forall_i} \text{int}(\Omega_i)$ is a union of open sets, it is therefore open itself. Thus, all evaluations are computed on some open set, effectively removing boundaries, where problems might have otherwise occurred. Theorem follows directly from the proof of Theorem 4.1 through argument (22). □

Branching of programs into domains (50) is done through conditional statements. Each conditional causes a doubling of possible paths.

Proposition 5.2. *Cardinality of the set of domains $\Omega = \{\Omega_i\}$ equals $|\{\Omega_i\}| = 2^k$, where k is the number of branching point within the program.*

This section concerns itself with employing the derived theorems to propose a treatment of branchings that is *linear in its complexity* and avoids the exponential threat of Proposition 5.2.

Theorem 5.6. *A program $P \in \mathcal{P}$ can be equivalently represented with at most $2n+1$ applications of the operator e^{∂}, on $2n+1$ analytic programs, where n is the number of branching points within the program.*

Proof. Source code of a program $P \in \mathcal{P}$ can be represented by a directed graph, as shown in Figure 1. Each branching causes a split in the execution tree, that after completion returns to the splitting point. By Theorem 5.3, each of these branches can be viewed as a program p_i, for which it holds

$$e^{\partial}(p_n \circ p_{n-1} \circ \cdots \circ p_1) = e^{\partial}(p_n) \circ e^{\partial}(p_{n-1}) \circ \cdots \circ e^{\partial}(p_1)$$

by Theorem 5.3.

Thus, the source code contains $2n$ differentiable branches, from its' first branching on, not counting the branch leading up to it, upon which the application of the operator e^{∂} is needed. Total of $2n+1$. By Theorem 4.1, each of these branches is analytic. □

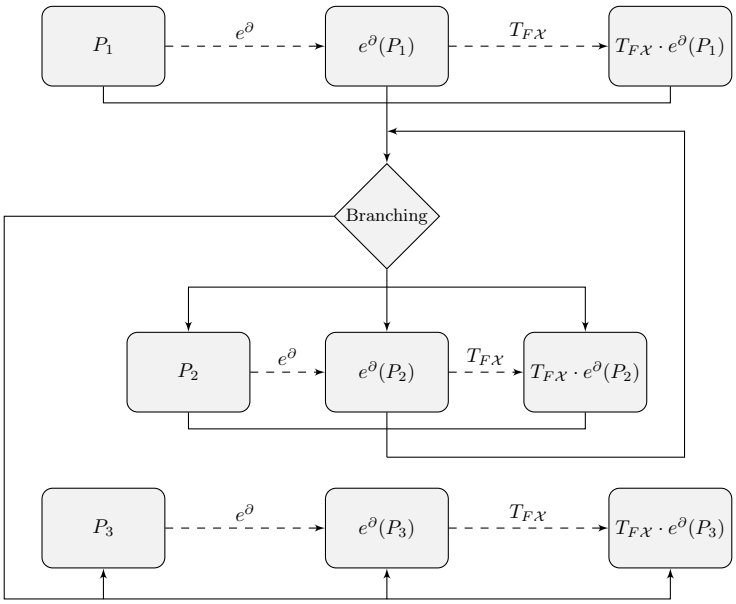

Figure 1: Transformation diagram

References

[1] John Backus. *Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs.* ACM, 2007.

[2] Andreas Blass. "Seven Trees in One". In: *arXiv:math/9405205* (1994).

[3] Marcelo Fiore and Tom Leinster. "Objects of Categories as Complex Numbers". In: *Advances in Mathematics 190 (2005), 264-277* (2002).

[4] Vito Volterra. *Theory of Functionals and of Integral and Integro-Differential Equations.* Dover Publications, 2005. ISBN: 0486442845.

[5] Norbert Wiener. "The operational calculus". In: *Mathematische Annalen* 95 (1926), pp. 557–584.

[6] Kamil A. Khan and Paul I. Barton. "A vector forward mode of automatic differentiation for generalized derivative evaluation". In: *Optimization Methods and Software* 30.6 (2015), pp. 1185–1212.

[7] Robin J. Hogan. "Fast Reverse-Mode Automatic Differentiation Using Expression Templates in C++". In: *ACM Trans. Math. Softw.* 40.4 (2014), 26:1–26:16. ISSN: 0098-3500.

[8] Mark Girolami and Ben Calderhead. "Riemann manifold Langevin and Hamiltonian Monte Carlo methods". In: *Journal of the Royal Statistical Society* (2011).

[9] Barak A. Pearlmutter and Jeffrey M Siskind. "Putting the Automatic Back into AD: Part I, What's Wrong (CVS: 1.1)". In: *ECE Technical Reports.* (2008).

[10] Barak A. Pearlmutter and Jeffrey M Siskind. "Putting the Automatic Back into AD: Part II, Dynamic, Automatic, Nestable, and Fast (CVS: 1.1)". In: *ECE Technical Reports.* (May 2008).