Operational Calculus for Differentiable Programming

Žiga Sajovic[1] and Martin Vuk [2]

[1]XLAB d.o.o.
ziga.sajovic@xlab.si
[2]University of Ljubljana, Faculty of Computer and Information Science
martin.vuk@fri.uni-lj.si

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 by way of *operational calculus*.

To this purpose, we develop an *abstract computational model of automatically differentiable programs* of arbitrary order. In the model, programs are elements of *programming spaces* and are viewed as maps from the *virtual memory space* to itself. Virtual memory space is 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 differentiable composition in the language. We prove that our language enables differentiable derivatives of programs by the use of the *order reduction map*. We demonstrate our models algebraic power over analytic properties of differentiable programs by analysing iterators, considering *fractional iterations* and their *iterating velocities*. We than solve the special case of *ReduceSum*.

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

1

programming, where a mapping has been shown between grammars and semirings [2]. Yet due to the lack of inverses (hence the semiring structure) they remain limited in the algebraic manipulations they 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 development 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 meaningful algebraic structure over differentiable programs. As such, there is still a need for a framework that properly captures the analytic properties of differentiable programs and provides higher-order constructs that can reason about them.

The ideas of functional programming and automatic differentiation have been combined to some extent successfully in the field of Deep Learning for example. It has shown itself to be more than a collection of machine learning algorithms and the name *Differentiable Programming* emerged 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 an algebraic language.

We have been inspired by the development of differentiable programming to formalize a theoretical model, that encompasses the ideas underlying differentiable programming and provides a more general setting for investigations of differentiable programs. The presented theoretical model enables analytic investigations of differentiable programs through algebraic tools, that are closer to the field of programming; i.e. the presented operators can take the same role as higher order functions in functional programming. We introduce a *Virtual Tensor Machine* as a language that extends functional definition of programs with a *Tensor Series Algebra* of the memory. Such a tensor description of the memory can also serve as a formalization of recent advancements in high performance computing hardware, ex. tensor processing units by Google and tensor cores by Nvidia. This algebraic structure inherent to our model allows us to establish an *Operational Calculus* of higher-order constructs that can facilitate reasoning about differentiable programs. Furthermore, the presented model is self-sufficient, as the Operational Calculus presented herein is implemented strictly within the language itself.

Mathematical analysis and calculus found its way into programming, where different fields employ analytic properties of programs. What seems to be lacking in these attempts is a mechanism that would abstract away the gory details of calculus and provide the ability to employ it on a higher level in computer programming. The proposed theoretical model and the constructed operational calculus aim to fill this gap. To demonstrate its algebraic power over analytic properties of differentiable programs, we analyse iterators, considering fractional iterations and their (higher-order) rates of change in relation to the number of iterations; i.e. (higher-order) *iterating velocities* . We than solve the special case of *ReduceSum* and its (higher-order) iterating velocities, and provide their explicit solutions within the language.

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 a 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 the* 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, 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

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

---

[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 $\mathbb{R}$

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 the vector space V to the 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 derivative of a computer program in an *Euclidean machine* as a program in the same *Euclidean machine*. We define three subspaces of the 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

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

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 the 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 added structure, this is precisely what to focus on. Loosely speaking, functional programming is described by monoids, and as such a tensor algebraic description of the memory space is 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

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

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, an arbitrary element of the memory space $\mathbf{W} \in \mathcal{V}_n$ is a mapping

$$\mathbf{W} : \mathcal{V} \to \mathcal{V}, \tag{15}$$

defined as

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

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*

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

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.

---

[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.

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 subspaces 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.

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 $P_{\alpha,k}^i$ 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 $\mathcal{V}$.

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

$$\implies$$

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

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

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:*

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

---

[4]The operator ∂ may be defined partially for other maps as well.

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] We will call (24) the tensor series algebra of the programming space.

4.3   Virtual Tensor Machine

We propose an abstract computational model that is capable of constructing differentiable programming spaces and provides a framework for algebraic study of analytic properties of differentiable programs.

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 the tensor series algebra of the programming space $\mathcal{P}_0 \otimes T(\mathcal{V}^*)$. 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*

- *$\mathcal{V}$ 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

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

---

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

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 $\mathbf{x}$:

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

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). $\qquad\square$

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

With the fundamentals of our framework established, we can begin to implement operators within its algebra. We begin by implementing an operator that shifts the program from its initial value and can later be used for the implementation of iterators and composers.

In the 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

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

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

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

in which partial applications can be made complete

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

by taking the image of the map $e^{h\partial}(P)$ at a certain point $\mathbf{v} \in \mathcal{V}$. Thus, we 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

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

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 (by switching the order of partial applications in (32)).

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 $\mathbf{u} \otimes f_1 \otimes \ldots \otimes f_n \in \mathcal{V} \otimes (\mathcal{V}^*)^{\otimes n}$ the contraction by $\mathbf{v} \in \mathcal{V}$ is given by applying co-vector f_n to $\mathbf{v}$ [6]

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

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

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

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

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

---

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

[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.

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*

$$P(\mathbf{v}_0 + h\mathbf{v}) = \left((e^{h\partial}P)(\mathbf{v}_0) \right)(\mathbf{v}) = \sum_{n=0}^{\infty} \frac{h^n}{n!} \partial^n P(\mathbf{v}_0) \cdot (\mathbf{v}^{\otimes n})$$

$$= \sum_{n=0}^{\infty} \frac{h^n}{n!} \sum_{\forall i,\alpha} \frac{\partial^n P_i}{\partial x_{\alpha_1} \ldots \partial x_{\alpha_n}} \mathbf{e}_i \cdot dx_{\alpha_1}(\mathbf{v}) \cdot \ldots \cdot dx_{\alpha_n}(\mathbf{v}). \quad (36)$$

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.
$\Longrightarrow$

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

$\Longleftarrow$

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

$$\wedge$$

$$\left. \partial^n e^{h\partial} \right|_{h=0} = \left. \sum_{i=0}^{\infty} \frac{h^i \partial^{i+n}}{i!} \right|_{h=0} = \partial^n$$

$$\Longrightarrow$$

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

$\square$

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^{∂} is contained strictly within the language

$$e^{h\partial}(\mathcal{P}_0) \subset \mathcal{P}_0 \otimes \mathcal{T}(\mathcal{V}^*), \quad (37)$$

and that the operator $e^{h\partial}$ is an automorphism of the programming algebra $\mathcal{P}_{\infty}$,

$$e^{h\partial}(p_1 \cdot p_2) = e^{h\partial}(p_1) \cdot e^{h\partial}(p_2) \quad (38)$$

where $\cdot$ stands for any bilinear map.

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 language, 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. Upon completion we will demonstrate how to perform calculations on the operator level, before they are applied to a particular programming space, which serves as a level of abstraction over the tensor series algebra of the memory space.

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

$$e^{h\partial}(f \circ g) = \exp(\partial_f e^{h\partial_g})(g, f) \tag{39}$$

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.
$\Longrightarrow$

$$\lim_{\|h\| \to 0} (\frac{d}{dh})^n e^{\partial}(f \circ g) = \lim_{\|h\| \to 0} \partial^n e^{h\partial}(f \circ g)$$
$$\Longrightarrow$$
$$\partial^n(f \circ g) \tag{40}$$

$\Longleftarrow$

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

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

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

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 (41) equals (40) by Faà di Bruno's formula.

$$\lim_{\|h\| \to 0} (\frac{d}{dh})^n \exp(\partial_f e^{h\partial_g}) = \sum_{\lambda(n)} n! \prod_{k \cdot l \in \lambda} \left(\frac{\partial_f \partial_g^l(g(v))}{l!}\right)^k \frac{1}{k!} \left(f(g(\mathbf{v}))\right) \tag{42}$$

$\square$

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 (39) and (41).

The operator of program composition

$$\exp(\partial_f e^{h\partial_g}) : \mathcal{P} \to \mathcal{P} \to \mathcal{P}_\infty, \tag{43}$$

allows two kinds of partial applications. The operator resulting from fixing the second map g in (43),

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

is the pullback of the generalized shift operator $e^{h\partial}$ through g. While the operator resulting from fixing the first map f in (43),

$$\exp(\partial_f e^{h\partial_g})(f, \cdot) = f_* \left(e^{h\partial} \right) \tag{45}$$

is the push-forward of the generalized shift operator $e^{h\partial}$ through f. This also generalizes the *U combinator* to its forward and backward modes, by restricting the *composers* (43) 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 $P = P_n \circ \ldots P_1$. 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 over the tensor series algebra of the memory space.

The derivative $\frac{d}{dh}$ of (44) is

$$\frac{d}{dh} \exp(\partial_f e^{h\partial_g})(g) = \partial_f (\partial_g g) e^{h\partial_g} \exp(\partial_f e^{h\partial_g})(g). \tag{46}$$

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

$$\frac{d}{dh} e^{h\partial_g} = \partial_g e^{h\partial_g}. \tag{47}$$

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* (39)

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

which is by equations (46) and (47), using algebra and correct applications equal to

$$\left(\partial_f(\partial_g^2 g)\right)e^{h\partial_g}\exp(\partial_f e^{h\partial_g})(g) + (\partial_f^2(\partial_g g)^2)e^{2h\partial_g}\exp(\partial_f e^{h\partial_g})(g) \qquad (48)$$

The operator is always shifted to the evaluating point (31) $\mathbf{v} \in \mathcal{V}$, thus, only the behaviour in the limit as $h \to 0$ is of importance. Taking this limit in the expression (48) 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 (41) 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 (38) and the derivative of the general shift operator (47). 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

$$\partial^n(f \circ g) = \left(\frac{d}{dh}\right)^n \exp\left(\partial_f e^{h\partial_g}\right)(g, f)\Big|_{h=0}. \qquad (49)$$

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 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*

$$\begin{array}{ccc} \mathcal{P}_n & \xrightarrow{\phi} & \mathcal{P}_{n-1} \\ \downarrow{\partial} & & \downarrow{\partial} \\ \mathcal{P}_{n+1} & \xrightarrow{\phi} & \mathcal{P}_n \end{array} \qquad (50)$$

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 writing of differentiable programs that act on derivatives of other programs is well defined within the language. This is a crucial feature, as stressed by other authors [9, 10]. 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   Iterators and Iterating Velocity

Iterator is an operator, composing a program $p : \mathcal{V} \to \mathcal{V} \in \mathcal{P}$ with itself. For ease of expression, we denote the n-th iterate of a program $p \in \mathcal{P}$, as p^n, as it is possible to view iteration as compositional exponentiation. In this view, one may seek to explore the relation between the value of the n-th iterate and n. With a form which expresses the iterate as a function of n, one could inquire its rate of change in relation to it and investigate fractional iterations, akin to fractional powers of other operators [11].

Let $\mathcal{I}_p$ be the monoid under composition $\circ$

$$\mathcal{I}_p = \{p^n : \mathcal{V} \to \mathcal{V}; \quad p(\mathbf{v}_f) = \mathbf{v}_f\}, \tag{51}$$

generated by $p \in \mathcal{P} : \mathcal{V} \to \mathcal{V}$ with some fixed point $\mathbf{v}_f \in \mathcal{V}$; note that any terminating program has a fixed point. We than turn towards analysing the structure of (51) in relation to n, the number of iterations.

Let $\mathcal{C}_p : h \to h \circ p$ be the operator of composition with p and assume h to be the solution of the eigen equation

$$\mathcal{C}_p(h) = \Lambda \cdot h \iff h(p(\mathbf{v})) = \Lambda \cdot h(\mathbf{v}). \tag{52}$$

It is clear that the composition of h with p^n is such,

$$h(p^n(\mathbf{v})) = \Lambda^n \cdot h(\mathbf{v}) \tag{53}$$

that in the image of h, iterations of p become multiplication with the eigen matrix Λ. This allows us to generalize the notion of an iteration from the integers, $n \in \mathbb{N}$, to the reals, $x \in \mathbb{R}$, and consider fractional iterations by

$$p^x(\mathbf{v}) = h^{-1}(\Lambda^x \cdot h(\mathbf{v})), \tag{54}$$

assuming h^{-1} exists. Note that we can express the eigen matrix Λ by differentiating (52) at the fixed point $\mathbf{v}_f$,

$$\partial h(p(\mathbf{v}_f)) \cdot \partial p(\mathbf{v}_f) = \Lambda \cdot \partial h(\mathbf{v}_f) \implies \partial p(\mathbf{v}_f) = \Lambda.$$

With foundations established, we can proceed to inquire about the rate of change of the values of a program p in relation to n, the number of iterations. Lets define the *rate of change operator* Ψ

$$\Psi : p^n \to \partial_n p^n, \tag{55}$$

that maps an iterate p^n to its *iterating velocities* $\partial_n p^n$. Of course

$$\Psi\left(p^n\right)\left(\mathbf{v}_f\right) = 0, \tag{56}$$

the iterating velocities of any iterate at the fixed point $\mathbf{v}_f$ are constantly zero, which is deduced from the (52) and reassures our intuition. Next, we introduce a change of variables $\Lambda = e^\nu$ for mathematical convenience and proceed towards computing the iterating velocity.

$$\partial_n h\left(p^n\right) = \partial_n\left(e^{\nu n} \cdot h\right)$$

$$\implies$$

$$\partial h\left(p^n\right) \cdot \partial_n p^n = \nu \cdot e^{\nu n} \cdot h \wedge e^{\nu n} \cdot h = h\left(p^n\right)$$

$$\implies$$

$$\Psi = \nu \cdot (\partial h)^{-1} \cdot h \tag{57}$$

The rate of change operator Ψ and iterating velocities [8] (57) can be used to study the importance of future iterations for the accumulating result, and aid with decisions on early stopping. Furthermore, the above indicates a subclass of programs for which the Halting problem can be analysed through the study of convergence and fixed points.

The computation of the eigen map h (52) was solved by Bridges [12] for any p with a power series representation. This result is extended to tensor series by the isomorphism to their quotient. Hence, as we can expand any $p \in \mathcal{P}$ into a tensor series by the use of the operator e^∂, by Theorem 5.2 the result also holds for any $p \in \mathcal{P}$.

5.6 ReduceSum in the Language of Operational Calculus

As a demonstration of the algebraic power over analytic conclusions inherent to our model, we examine the functional *ReduceSum*, and derive its explicit form as a function of n, the number of its iterations, or upper bound, with special interest in the rate of change of the functional in relation to n; i.e. iterating velocity and its higher order counter parts (acceleration etc.).

Let $\mathcal{S}^n$ denote the operator, that performs a linear shift of a program p in the direction $\mathbf{v}$, from its initial point $\mathbf{v}_0$. By Theorem 5.2 we have

$$\left(e^{n\partial}\big|_{\mathbf{v}_0} p\right)(\mathbf{v}) = p(\mathbf{v}_0 + n\mathbf{v}) \implies \mathcal{S}^n = e^{n\partial}\big|_{\mathbf{v}_0}, \tag{58}$$

---

[8]Higher derivatives can be derived by induction.

and thus clearly $\mathcal{S}^n \cdot \mathcal{S}^m = \mathcal{S}^{n+m}$ and $(\mathcal{S}^n + \mathcal{S}^m)(p) = \mathcal{S}^n(p) + \mathcal{S}^m(p)$, which we use to define the n-th reduction as

$$\mathcal{R}_+^n = (1 + \mathcal{S} + \mathcal{S}^2 + \cdots + \mathcal{S}^n),$$

that results in

$$\mathcal{R}_+^n(p)(\mathbf{v}) = \sum_{h=0}^{n} p(\mathbf{v}_0 + h\mathbf{v})$$

upon application.

With this we turn towards computing with operators alone to harness the algebraic power of our framework. We write

$$(1 + \mathcal{S} + \mathcal{S}^2 + \cdots + \mathcal{S}^n) = 1 + \mathcal{S}(1 + \mathcal{S} + \mathcal{S}^2 + \cdots + \mathcal{S}^{n-1})$$

$$\implies$$

$$1 - \mathcal{S}^n = (1 - S)\,\mathcal{R}_+^{n-1}$$

$$\implies$$

$$\mathcal{R}_+^{n-1} = (1 - \mathcal{S}^n)\left(\frac{1}{1 - \mathcal{S}}\right), \tag{59}$$

where $\frac{1}{1-\mathcal{S}}$ is to be understood in the sense of formal tensor series. We will denote $(1 - \mathcal{S}^n)$ by $\left[\,\cdot\,\right]_{\mathbf{v}_0}^n$, recognizing that it represents the action of shifting the program in the direction of $\mathbf{v}$ by a fraction of n, from its initial position $\mathbf{v}_0$, and subtracting the two; while $\mathbf{v}$ is yet to be applied. Taking Theorem 5.2 into account we write

$$\mathcal{R}_+^{n-1} = \left[\partial^{-1}\left(\frac{\partial}{1 - e^\partial}\right)\right]_{\mathbf{v}_0}^n. \tag{60}$$

Note that while ∂^{-1} is undetermined, its composition with $\left[\,\cdot\,\right]_{\mathbf{v}_0}^n$ is well defined on $\mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*)$. Also note, that the parenthesised expression, $\frac{\partial}{1-e^\partial}$, and ∂^{-1} are algebraic encodings of higher-order programs, which are to be expanded into explicit form upon application inside the tensor series algebra of the programming space $\mathcal{P} \otimes \mathcal{T}(\mathcal{V}^*)$. Doing so, we recognize

$$\frac{h\partial}{1 - e^{h\partial}} = \sum_{i=0}^{\infty} B_i \frac{(h\partial)^i}{i!} \tag{61}$$

B_i to be the n-th Bernoulli number. Thus the higher order program

$$\mathcal{R}_+^{n-1} : \mathcal{P} \to \left\{\mathcal{V} \to \mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*)\right\}$$

is expressed as

$$\mathcal{R}_+^{n-1} = \left[B_0\partial^{-1} + \sum_{i=1}^{\infty} B_i \frac{\partial^{i-1}}{i!}\right]_{\mathbf{v}_0}^n. \tag{62}$$

17

Upon applying it to a program at a particular point $\mathbf{v} \in \mathcal{V}$ this becomes

$$\mathcal{R}_+^{n-1} p(\mathbf{v}) = \left[B_0 \partial^{-1} p(t) + \sum_{i=1}^{\infty} B_i \frac{\partial^{i-1} p(t)}{i!} \right]_{\mathbf{v}_0}^{n\mathbf{v}} (\mathbf{v}), \qquad (63)$$

where the evaluation at $\mathbf{v} \in \mathcal{V}$ as by (35), performs the needed translation [9], as the image of the operator $\mathcal{R}_+^{n-1}$ is an element of the tensor series algebra of the memory space $\mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*)$. Note that n, the number of iterations, is the only remaining free variable, as desired.

Remark 5.5. *When $p \in \mathcal{P}$ is an univariate mapping, the expression* (63) *recovers the Euler-Maclaurin integral formula [13], which is demonstrated by applying the operator* (62) *to the function x^m*

$$\mathcal{R}_+^n (x^m) = \frac{1}{m+1} \sum_{i=0}^{m} \binom{m+1}{i} B_i \cdot n^{m+1-i},$$

and producing the closed form solution.

Furthermore, due to the operational algebra of higher order programs established by our model, we can compute the operator of iterating velocity (and higher order change) of the n-th iterate by differentiating the operator $\mathcal{R}_+^{n-1}$ itself. Reverting to the form (60) and substituting (58), we have

$$\frac{d^k}{dn^k} \mathcal{R}_+^{n-1} = \frac{d^k}{dn^k} \left((1 - e^{n\partial}) \left(\frac{1}{1 - e^{\partial}} \right) \right) = \mathcal{S}^n \left(\frac{\partial^n}{1 - e^{\partial}} \right), \qquad (64)$$

where commutativity of shifting and differentiating was used. Noting that $\mathcal{S}^n$ simply shifts the operand in the direction of $\mathbf{v}$ by a factor of n, the explicit form

$$\frac{d^k}{dn^k} \mathcal{R}_+^{n-1} \mathcal{P}|_{n=N} (\mathbf{v}) = \left(\sum_{i=0}^{\infty} B_i \frac{\partial^{N-1+i} \mathcal{P} (\mathbf{v}_0 + N\mathbf{v})}{i!} \right) (\mathbf{v}) \qquad (65)$$

where the evaluation at $\mathbf{v} \in \mathcal{V}$ once again performs the needed translation.

6 Conclusions

In this paper we presented a theoretical model for differentiable programming. Throughout the course of the paper we have shown the model to be a complete description of differentiable programming. Furthermore, the innate algebraic structure of the framework supplements the descriptive power of a language with the ability to reason about the programs it implements, by way of operational calculus. We believe operational calculus has a place in the evolution of computer science, where languages are to be endowed with algebraic constructs that hold power over analytic properties of the programs they implement. These results hope to inspire other practitioners of differentiable programming to reach for operational calculus on their quest to further the field.

---

[9]Both the translation and the shift are to be performed (at) by the same point $\mathbf{v} \in \mathcal{V}$.

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).

[11] Hikosaburo Komatsu. "Fractional powers of operators". In: *Pacific Journal of Mathematics* 19.2 (1966), pp. 285–346.

[12] Robert A Bridges. "A solution to Schröder's equation in several variables". In: *Journal of Functional Analysis* 270.9 (2016), pp. 3137–3172.

[13] Tom M Apostol. "An elementary view of Euler's summation formula". In: *The American Mathematical Monthly* 106.5 (1999), pp. 409–418.