Operational Calculus on Programming Spaces

Žiga Sajovic (corresponding author) ZIGA.SAJOVIC@GMAIL.COM
University of Ljubljana, Faculty of Computer and Information Science, Večna pot 113, SI-1000 Ljubljana, Slovenia
XLAB d.o.o., Pot za Brdom 100, SI-1000 Ljubljana, Slovenia

Martin Vuk MARTIN.VUK@FRI.UNI-LJ.SI
University of Ljubljana, Faculty of Computer and Information Science, Večna pot 113, SI-1000 Ljubljana, Slovenia

Editor:

Abstract

In this paper we develop operational calculus on programming spaces that generalizes existing approaches to automatic differentiation of computer programs and provides a rigorous framework for program analysis through calculus.

We present an abstract computing machine that models automatically differentiable computer programs. Computer programs are viewed as maps on a finite dimensional vector space called virtual memory space, which we extend by the tensor algebra of its dual to accommodate derivatives. The extended virtual memory is by itself an algebra of programs, a data structure one can calculate with, and its elements give the expansion of the original program as an infinite tensor series at program's input values. We define the operator of differentiation on programming spaces and implement a generalized shift operator in terms of its powers. Our approach offers a powerful tool for program analysis and approximation, and provides deep learning with a formal calculus.

Such a calculus connects general programs with deep learning, through operators that map both formulations to the same space. This equivalence enables a generalization of the existing methods for neural analysis to any computer program, and vice versa. Several applications are presented, most notably a meaningful way of neural network initialization that leads to a process of program boosting.

Keywords: programming spaces, operational calculus, deep learning, neural networks, differentiable programs, tensor calculus, program analysis

1. Introduction

Programming holds the power of algorithmic control flow and freedom of expression, whose abstinence severely limits descriptiveness of closed form methods of *pen and paper* mathematics, thus firmly cementing programming as the language of modern problem solving. Yet, a vibrant tradition of mathematics has existed since the dawn of mind, that remains, with the exception of combinatorics, largely untapped by computer science.

Just as the discrete nature of physical reality is studied through analytic means, so can the nature of digital phenomena. Studying these procedures as objects undergoing change in some virtual space, has partially been attempted in some fields, such as Hamiltonian Monte

Carlo methods of Bayesian predictors, that Girolami and Calderhead (2011) studied as manifolds to great success, using unfortunately impractical methods of hard-coding derivatives of distributions. This of course stifles the freedom of algorithmic expression programming is embraced for.

The way evaluation of algorithmic expressions differs from evaluation of symbolic expressions of standard analysis, lies at the very core of this dissonance. The disconnect was partially mitigated by methods of automatic differentiation utilized today in machine learning, engineering, simulations, etc. (see Baydin and et. al., 2015). Yet under the lack of a proper formalism the model collapses (see Pearlmutter and Siskind, 2008b) when one tries to generalize to such notions as a differentiable program p_1 operating on (differentiable) derivatives of another program p_2 (where only coding of p_2 is required within p_1), while retaining the famed expressive freedom. Models allowing for nested differentiation as in Pearlmutter and Siskind (2008a), still fail in providing algorithms with an algebra enabling study and formulation of programs with analytic equations. Existing models remain nothing more than efficient means to calculating derivatives, void of any meaningful algebraic insight, lacking the true power the vast field of analysis is endowed with.

The aim of this paper is bridging the gap between programming and analysis left behind by previous models. By generalizing them, they are shown to be specific views of the same great elephant. Employing tensor algebra a virtual memory is constructed, whose data structures one can calculate with. In Section 5, an exact definition and construction of an analytic virtual machine is presented, capable of implementing infinitely-differentiable programming spaces and operators acting upon them, supporting actions on multiple memory locations at a time. These programming spaces are shown to be a subalgebra, giving rise to symbolic manipulations of programs, while attaining construction by algorithmic control flow. Operators expanding a program into an infinite tensor series are derived in Section 5.2. The operator of program composition is constructed in Section 5.3, generalizing both forward (e.g., Khan and Barton, 2015) and reverse (e.g., Hogan, 2014) mode of automatic differentiation to arbitrary order under a single invariant operator. The problem of nested differentiation is resolved in Section 5.5.

Theory grants us the ability to choose programs' complexity through approximations in the virtual space. The tensor algebra model of virtual memory consists of multi-linear maps which are tailor made for efficient implementation by GPU parallelism (Abdelfattah and et al., 2016). In Section 5.6 functional transformations of programs in an arbitrary function basis are derived. As different hardware is optimized for running with different sets of functions, this proves useful with adapting code to fit specific hardware. Branching and discontinuity are discussed in Section 5.8. where we show that analytic virtual machines fully integrate control structures, retaining algorithmic control flow.

Theory grants a generalization from neural networks to general tensor networks in Section 6, revealing their connection with programs in Section 6.1. The historic Taylor series are modernized in Section 6.1.1 by deriving *Neural tensor series*, establishing an equivalence between general tensor networks and programming spaces. This has direct implications to the study and understanding of *differentiable neural computers* (Graves and et al., 2016) through operational calculus. By this equivalence, transformations of arbitrary programs to general tensor networks are derived. The transformation to a *Neural Tensor series* serves as a great initialization point of a general tensor network to be trained. This can be seen as a process of *boosting* (Freund and et al., 1999), converting a weaker learner to a strong one, which proves fruitful, as currently neural networks give best results to many problems. All such constructs are trainable (as is their training process itself) and adhere to the operational calculus. This is demonstrated in Section 6.2, enabling analytic study through the theory.

Theory offers new insights into programs, as we demonstrate how to probe their inner structure in Section 7, revealing what actions properties of the domain most react to in Section 7.1. This enables us to alter data by imposing some property it lacks upon it. State of the art methods for analyzing neural networks (Mordvintsev et al., 2015) are generalized for the use on programming spaces in Section 7.1.1, providing a framework for analysis of machine learning architectures and other virtual constructs, as actions on the virtual space.

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 virtual 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 1 (Euclidean virtual machine) *The tuple $(\mathcal{V}, \mathcal{F})$ is an Euclidean virtual 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.*

3. Differentiable Maps and Programs

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

Definition 2 (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 $\mathbf{x}$.*

[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 $\mathbb{R}$

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 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

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

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 (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*

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

Remark 4 *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 virtual machine again as a program in the same euclidean virtual 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 5 *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 subspaces spanned by the standard basis vectors, such that the map P_e, defined by the following commutative diagram*

$$\begin{tikzcd}
\mathcal{V} \arrow[r, "P"] & \mathcal{V} \arrow[d, "\mathrm{pr}_{O_P}"] \\
I_P \arrow[u, hook, "\vec{i} \mapsto \vec{i} + \vec{f}"] \arrow[r, "P_e"] & O_P
\end{tikzcd} \tag{9}$$

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 6 (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

Motivated by the Definition 6, we define virtual memory for differentiable programs as a sequence of vector spaces with the recursive formula

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

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

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

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 $\mathcal{V} \to \mathcal{V} \otimes T(\mathcal{V}^*)$. Putting memory considerations aside, we propose an universal model of the memory for differentiable programs.

Definition 7 (Virtual memory) *Let $(\mathcal{V}, \mathcal{F})$ be an Euclidean virtual machine and let*

$$\mathcal{V}_\infty = \mathcal{V} \otimes T(\mathcal{V}^*) = \mathcal{V} \oplus (\mathcal{V} \otimes \mathcal{V}^*) \oplus \dots, \tag{15}$$

where $T(\mathcal{V}^*)$ is the tensor algebra of the dual space $\mathcal{V}^*$. We call $\mathcal{V}_\infty$ the differentiable virtual memory *of a virtual computing 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 (15) 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 (15) to $\partial^k P$ and all others to zero.

4.1 Differentiable Programming Spaces

Let us define the following function spaces:

$$\mathcal{F}_n = \{ f : \mathcal{V} \to \mathcal{V} \otimes T_n(\mathcal{V}^*) \} \tag{16}$$

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 (16)

$$\partial^k : \mathcal{F}^n \to \mathcal{F}^{n+k}. \tag{17}$$

Definition 8 (Differentiable programming space) *A* differentiable programming space *$\mathcal{P}_0$ is any subspace of $\mathcal{F}_0$ such that*

$$\partial \mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(\mathcal{V}^*) \tag{18}$$

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*.*

[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] 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 9 (Infinite differentiability) *Any differentiable programming space $\mathcal{P}_0$ is an infinitely differentiable programming space, meaning that*

$$\partial^k \mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(\mathcal{V}^*) \tag{19}$$

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 $\mathcal{V}$.

$$\partial^{n+1} P^i_{\alpha,k} = \partial(\partial^n P^i_\alpha)_k \wedge (\partial^n P^i_\alpha) \in \mathcal{P}_0 \implies \partial(\partial^n P^i_\alpha)_k \in \mathcal{P}_0 \otimes T(\mathcal{V}^*) \tag{20}$$

$$\implies$$

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

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

Corollary 10 *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:*

$$\mathcal{P}_n < \mathcal{P}_0 \otimes T_n(\mathcal{V}^*). \tag{21}$$

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

$$\mathcal{P}_\infty < \mathcal{P}_0 \otimes \mathcal{T}(\mathcal{V}^*), \tag{22}$$

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]

5. Operational Calculus on Programming Spaces

By Corollary 10 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

$$\tau_n = 1 + \partial + \partial^2 + \ldots + \partial^n \tag{23}$$

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 $\mathbf{x}$:

$$\tau_k P(\mathbf{x}) = P(\mathbf{x}) + \partial_{\mathbf{x}} P(\mathbf{x}) + \partial^2_{\mathbf{x}} P(\mathbf{x}) + \ldots + \partial^k_{\mathbf{x}} P(\mathbf{x}). \tag{24}$$

The operator τ_n satisfies the recursive relation:

$$\tau_{k+1} = 1 + \partial \tau_k, \tag{25}$$

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

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

Proposition 11 *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 (20) of the proof of Theorem 9, allowing simple implementation, as dictated by (25). ■

Remark 12 *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 13 (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:*

$$(\mathbf{v} \otimes f_1 \otimes \dots f_k) \cdot (\mathbf{u} \otimes g_1 \otimes \dots g_l) \quad = \quad (\mathbf{v} \cdot \mathbf{u}) \otimes f_1 \otimes \dots f_k \otimes g_1 \otimes \dots g_l \tag{26}$$

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

Theorem 14 (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* (26)*.*

5.1 Analytic Virtual Machine

We propose an abstract computational model, a virtual machine capable of constructing differentiable programming spaces. Such a machine provides a framework for analytic study of algorithmic procedures by algebraic means.

Definition 15 (Analytic virtual 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 $\mathcal{P}_0$ is a differentiable programming space, this defines an infinitely differentiable virtual machine.

Remark 16 *The tuple $(\mathcal{V}, \mathcal{P}_0)$ and the structure of the* tensor series algebra $\mathcal{T}(\mathcal{V}^*)$ *are sufficient to construct* infinitely differentiable programing spaces $\mathcal{P}_\infty$ *by linear combinations of elements of $\mathcal{P}_0 \otimes \mathcal{T}(\mathcal{V}^*)$.*

An illustrative example of the implementation of an analytic virtual machine is available on GitHub (Žiga Sajovic, 2016a). Implementation closely follows theorems and derivations of this paper and is intended as an educational guide for those transitioning from automatic differentiation to this theory. A paper (Žiga Sajovic, 2016b) explaining the process of implementation accompanies the source-code.

5.2 Tensor Series Expansion

There exists a space spanned by the set $\mathcal{D}^n = \{\partial^k; \quad 0 \leq k \leq n\}$ over a field K. Thus, the expression

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

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

$$e^{h\partial} = \sum_{n=0}^{\infty} \frac{h^n}{n!} \sum_{\forall_{i,\alpha}} \frac{\partial^n}{\partial x_{\alpha_1} \dots \partial x_{\alpha_n}} \mathbf{e}_i \otimes dx_{\alpha_1} \otimes \dots \otimes dx_{\alpha_n}. \tag{27}$$

The operator $e^{h\partial}$ is a mapping between function spaces (16)

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

It also defines a map

$$e^{h\partial} : \mathcal{P} \times \mathcal{V} \to \mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*), \tag{28}$$

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 (28). 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 (22), 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* $\mathcal{T}(\mathcal{V}^*)$, arriving at

$$e^{h\partial} : \mathcal{P} \times \mathcal{V} \to \mathcal{V} \otimes \mathcal{S}(\mathcal{V}^*) \tag{29}$$

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]

$$\mathbf{u} \otimes f_1 \otimes \dots \otimes f_n \cdot \mathbf{v} = f_n(\mathbf{v}) \mathbf{u} \otimes f_1 \otimes \dots f_{n-1}. \tag{30}$$

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

$$\mathbf{u} \otimes f_1 \otimes \dots \otimes f_n \cdot (\mathbf{v})^{\otimes n} = f_n(\mathbf{v}) f_{n-1}(\mathbf{v}) \cdots f_1(\mathbf{v}) \mathbf{u}, \tag{31}$$

Both contractions (30) and (31) 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 $\mathbf{M} = \mathbf{m}_0 + \mathbf{m}_1 + \dots + \mathbf{m}_n \in \mathcal{V} \otimes T_n(\mathcal{V}^*)$, where $\mathbf{m}_k \in$

[6] For order two tensors from $\mathcal{V} \otimes \mathcal{V}^*$ the contraction correspons 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 (31) to $\mathcal{V} \otimes T(\mathcal{V}^*)$ can be seen as a generalzation of the affine map, where the zero order tensors account for translation.

$\mathcal{V} \otimes (\mathcal{V}^*)^{\otimes k}$, applying the contraction by a vector $\mathbf{v} \in \mathcal{V}$ multiple times yields a polynomial map

$$\mathbf{M}(\mathbf{v}) = \mathbf{m}_0 + \mathbf{m}_1 \cdot \mathbf{v} + \ldots + \mathbf{m}_n \cdot (\mathbf{v})^{\otimes n}. \tag{32}$$

Theorem 17 *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*

$$\begin{aligned} P(\mathbf{v}_0 + h\mathbf{v}) = \Big((e^{h\partial}P)(\mathbf{v}_0)\Big)(\mathbf{v}) &= \sum_{n=0}^{\infty} \frac{h^n}{n!} \partial^n P(\mathbf{v}_0) \cdot (\mathbf{v}^{\otimes n}) \\ &= \sum_{n=0}^{\infty} \frac{h^n}{n!} \sum_{\forall_{i,\alpha}} \frac{\partial^n P_i}{\partial x_{\alpha_1} \ldots \partial x_{\alpha_n}} \mathbf{e}_i \cdot dx_{\alpha_1}(\mathbf{v}) \cdot \ldots \cdot dx_{\alpha_n}(\mathbf{v}). \end{aligned} \tag{33}$$

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.

$\Longrightarrow$

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

$\Longleftarrow$

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

$$\wedge$$

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

$$\Longrightarrow$$

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

■

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

$$e^{h\partial}(p_1 \cdot p_2) = e^{h\partial}(p_1) \cdot e^{h\partial}(p_2) \tag{34}$$

where $\cdot$ stands for any bilinear map.

Remark 18 (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 (Wiener, 1926). The theory presented in this section offers more than a mere shift, which will become apparent in the coming sections.*

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.

Remark 19 *Independence of the operator* (28) *from a coordinate system, translates to independence in execution. Thus the expression* (33) *is invariant to the point in execution of a program, a fact we explore in Section 5.8.*

Remark 20 *Corollary* 10 *through* (22) *implies* $e^{h\partial}(\mathcal{P}_0) \subset \mathcal{P}_0 \otimes \mathcal{T}(\mathcal{V}^*)$ *which enables efficient implementation by operator* τ.

5.3 Operator of Program Composition

In this section both forward (e.g, Khan and Barton, 2015) and reverse (e.g., Hogan, 2014) mode of automatic differentiation are generalized to arbitrary order under a single invariant operator in the theory. We demonstrate how to perform calculations on operator level before they are applied to a particular programming space, condensing complex notions to simple expressions.

Theorem 21 (Program composition) *Composition of maps* $\mathcal{P}$ *is expressed as*

$$e^{h\partial}(f \circ g) = \exp(\partial_f e^{h\partial_g})(g, f) \tag{35}$$

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.

$\Longrightarrow$

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

$$\Longrightarrow$$

$$\partial^n(f \circ g) \tag{36}$$

$\Longleftarrow$

$$\exp(\partial_f e^{h\partial_g}) = \exp\left(\partial_f \sum_{i=0}^{\infty} \frac{(h\partial_g)^i}{i!}\right) = \prod_{i=1}^{\infty} e^{\partial_f \frac{(h\partial_g)^i}{i!}} \left(e^{\partial_f}\right)$$

$$\Longrightarrow$$

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

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

$$\lim_{\|h\|\to 0} (\frac{d}{dh})^n \exp(\partial_f e^{h\partial_g}) = \sum_{\lambda(n)} n! \prod_{k \cdot l \in \lambda} \Big(\frac{\partial_f \partial_g^l(g)}{l!}\Big)^k \frac{1}{k!} \Big(\Big(e^{\partial_f}\Big) f\Big) \tag{37}$$

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 (37) equals (36) by Faà di Bruno's formula.

$$\lim_{\|h\|\to 0} (\frac{d}{dh})^n \exp(\partial_f e^{h\partial_g}) = \sum_{\lambda(n)} n! \prod_{k \cdot l \in \lambda} \Big(\frac{\partial_f \partial_g^l(g(v))}{l!}\Big)^k \frac{1}{k!} \Big(f(g(\mathbf{v}))\Big) \tag{38}$$

■

The Theorem 21 enables an invariant implementation of the operator of program composition in $\mathcal{P}_n$, expressed as a tensor series through (35) and (37).

By fixing the second map g in

$$\exp(\partial_f e^{h\partial_g}) : \mathcal{P} \times \mathcal{P} \to \mathcal{P}_\infty, \tag{39}$$

the operator

$$\exp(\partial_f e^{h\partial_g})(\cdot, g) = g^* \left(e^{h\partial}\right) \tag{40}$$

is the pullback through g of the generalized shift operator $e^{h\partial}$. While by fixing the first map f in (39), the operator

$$\exp(\partial_f e^{h\partial_g})(f, \cdot) = f_* \left(e^{h\partial}\right) \tag{41}$$

is the push-forward through f of the generalized shift operator $e^{h\partial}$.

Remark 22 (Unified AD) *If a program is written as $P = P_n \circ \dots P_1$, than 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. Both forward and reverse mode (generalized to arbitrary order) are obtainable using the same operator* (39)*, by fixing the appropriate map f or g. This generalizes both concepts under a single operator.*

Remark 23 *The operator* (35) *can be generalized for the notion of a pullback to arbitrary operators.*

Thus, through (35) and all its' descendants (exponents), the operator (40) grants invariance to the point in execution of a program, which is important when proving algorithm's correctness. This is analogous to the principle of general covariance (see section 7.1 in a book by O'Hanian et al., 1994) in general relativity, the invariance of the form of physical laws under arbitrary differentiable coordinate transformations.

Corollary 24 *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 (29) and Theorem 21. ■

Remark 25 *With explicit evaluations in Corollary 24*

$$e^{h\partial}|_{\mathbf{v}_0}(p_n \circ \cdots \circ p_0) = e^{h\partial}|_{\mathbf{v}_n}(p_n) \circ \cdots \circ e^{h\partial}|_{\mathbf{v}_0}(p_0)$$

the wise choice of evaluation points is $\mathbf{v}_i = p_{i-1}(\mathbf{v}_{i-1}) \in \mathcal{V}$.

With this we turn towards easing such calculations, completing them on the level of operators. The derivative $\frac{d}{dh}$ of (40) is

$$\frac{d}{dh}\exp(\partial_f e^{h\partial_g})(g) = \partial_f(\partial_g g)e^{h\partial_g}\exp(\partial_f e^{h\partial_g})(g) \tag{42}$$

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

$$\frac{d}{dh}e^{h\partial_g} = \partial_g e^{h\partial_g} \tag{43}$$

We may now compute derivatives (of arbitrary order) of the pullback operator.

5.4 Example of an Operator Level Computation

As an example we compute the second derivative.

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

which is by equations (42) and (43), using algebra and correct applications

$$\left(\partial_f(\partial_g^2 g)\right)e^{h\partial_g}\exp(\partial_f e^{h\partial_g})(g) + (\partial_f^2(\partial_g g)^2)e^{2h\partial_g}\exp(\partial_f e^{h\partial_g})(g) \tag{44}$$

The operator is always shifted to the evaluating point (28) $\mathbf{v} \in \mathcal{V}$, thus, only the behavior in the limit as $h \to 0$ is of importance. Taking this limit in the expression (44) 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 (37) for $n = 2$.

As it is evident from the example, calculations using operators are far simpler, than direct manipulations of functional series, similar to how it was done in the proof of Theorem 21. This enables a simpler implementation that functions over arbitrary programming (function) 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 (34), the derivative of the composition operator (42) and the derivative of the general shift operator (43). 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 composition can also be computed on the operator level:

$$\partial^n(f \circ g) = \left(\frac{d}{dh}\right)^n \exp\left(\partial_f e^{h\partial_g}\right)(g, f)\Big|_{h=0}. \tag{45}$$

5.5 Order Reduction for Nested Applications

It is useful to be able to use the k-th derivative of a program $P \in \mathcal{P}$ as part of a different differentiable program P_1. As such, we must be able to treat the derivative itself as a differentiable program $P'^k \in \mathcal{P}$, while only coding the original program P.

Theorem 26 (Order reduction) *There exists a reduction of order map $\phi : \mathcal{P}_n \to \mathcal{P}_{n-1}$, such that the following diagram commutes*

$$\begin{tikzcd}
\mathcal{P}_n \arrow[r, "\phi"] \arrow[d, "\partial"] & \mathcal{P}_{n-1} \arrow[d, "\partial"] \\
\mathcal{P}_{n+1} \arrow[r, "\phi"] & \mathcal{P}_n
\end{tikzcd} \tag{46}$$

satisfying

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

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

Corollary 27 (Differentiable derivative) *By Theorem 26, n-differentiable k-th derivatives of a program $P \in \mathcal{P}_0$ can be extracted by*

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

We gained the ability of writing a differentiable program acting on derivatives of another program, stressed as crucial (but lacking in most models) by Pearlmutter and Siskind (2008a). Usage of the reduction of order map and other constructs of this Section are demonstrated in Section 7.

5.6 Functional Transformations of Programs

Let's suppose a hardware H is optimized for the set of functions $F = \{f_i : \mathcal{V} \to \mathcal{V}\}$. The set F is specified by the manufacturer.

With technological advances, switching the hardware is common, which can lead to a decline in performance. Thus, we would like to employ transformations of a program $P \in \mathcal{P}$ in basis F. It is common to settle for a suboptimal algorithm, that is efficient on hardware H. Sub-optimality of the algorithm depends on the set F, whether it spans P or not. A classic example of a transformation of basis is the Fourier transform.

Using the developed tools, the problem is solvable using linear algebra. Let e_n^∂ denote the projection of the operator e^∂, onto the first n basis vectors $\{\partial^i\}$. By Theorem 17 a map (29) from the space of programs, to the space of polynomials, with unknowns in $\mathcal{V}^k$, can be constructed using the operator e^∂. Let $\mathcal{X} = \{p_i\}$ denote a basis of the space of polynomials $\mathcal{V} \to \mathcal{V}$ [8]. We can interpret $e_n^\partial(P \in \mathcal{P})$ as a vector of linear combinations of $\mathcal{X}$.

[8] One choice would be the monomial basis, consisting of elements $\mathbf{e}_i \otimes \prod_{\alpha, \forall_j} x_{\alpha_j}$, where $\mathbf{e}_i$ span $\mathcal{V}$, x_i span $\mathcal{V}^*$ and α multi-index

We define the tensor $T_{\mathcal{X}F}$ of basis transformation $F \to \mathcal{X}$ by

$$T_{\mathcal{X}F} = p_1 \otimes e_n^{\partial}(f_1)^* + p_2 \otimes e_n^{\partial}(f_2)^* + \ldots + p_n \otimes e_n^{\partial}(f_n)^*. \tag{47}$$

The tensor of basis transformation $\mathcal{X} \to F$ is the inverse

$$T_{F\mathcal{X}} = T_{\mathcal{X}F}^{-1}. \tag{48}$$

For a specific set F (and consequentially a hardware H, upon which the set F is conditioned), the tensor (48) only has to be computed once, and can then be used for transforming arbitrary programs (while using the same operator e_n^{∂}). The coordinates of program $P \in \mathcal{P}$ in basis F are

$$P_F = T_{F\mathcal{X}} \cdot e^{\partial}(P) \tag{49}$$

The expression (49) represents coordinates of program P in basis F. Thus, the program is expressible as a linear combination of f_i, with components P_F as coefficients.

$$P = \sum_{i=0}^{n} P_{Fi} f_i$$

If F does not span $\mathcal{P}$, or we used the projection of the operator $e_{n<N}^{\partial}$, the expression P_F still represents the best possible approximation of the original program, on components $\{\partial^n\}$, in basis F.

Remark 28 *It makes sense to expand the set F, by mutual (nested) compositions, and gain mappings before computing the tensor* (48) *and increase the power of the above method.*

5.7 Special Case of Functions $\mathcal{V} \to K$

We describe a special case when $\mathcal{P}_0 = \mathcal{V} \otimes \mathcal{P}_{-1}$ and $\mathcal{P}_{-1} < K^{\mathcal{V}}$ is a subspace of the space of functions $\mathcal{V} \to K$. This is useful, if the set F only contains functions $\mathcal{V} \to K$. It is very common, that basic operations in a programming language change one single real valued variable at a time. In that case, the value of changed variable is described by the function $\tilde{f} : \mathcal{V} \to K$, while the location, where the value is saved is given by a standard basis vector $\mathbf{e}_i$. The map $f : \mathcal{V} \to \mathcal{V}$ is then given as a tensor product $f = \mathbf{e}_i \otimes \tilde{f}$. We can start the construction of the differentiable programming spaces by defining differentiable programming space of functions $\mathcal{V} \to K$ instead of maps $\mathcal{V} \to \mathcal{V}$ as in definition 8. Analog to the the Theorem 9 and Corollary 10 it is easy to verify, that

$$\partial^k \mathcal{P}_{-1} < \mathcal{P}_{-1} \otimes T_k(\mathcal{V}^*). \tag{50}$$

Since tensoring with elements of $\mathcal{V}$ commutes with differentiation operator ∂

$$\partial^k \mathcal{P}_0 < \mathcal{V} \otimes \partial^k \mathcal{P}_{-1} \tag{51}$$

and analytic virtual machine can be defined in terms of functions $\mathcal{P}_{-1}$, enabling more efficient implementation of the operators ∂ and e^{∂}. The functional transformation becomes much more efficient, since the set of functions F can be generated by the functions of the form $f = \mathbf{e}_i \otimes \tilde{f}_j$, where $F_{-1} = \{\tilde{f}_j : \mathcal{V} \to K\}$.

Theorem 29 *Suppose that $\mathcal{P}_0 = \mathcal{V} \otimes \mathcal{P}_{-1}$ where $\mathcal{P}_{-1}$ is a subspace of functions $\mathcal{V} \to K$. Suppose that $F = \{\mathbf{e}_i \otimes \tilde{f}_j\}$ and $\mathcal{X} = \{\mathbf{e}_l \otimes \tilde{p_k}\}$ is the basis of the space of polynomial maps $\mathcal{V} \to \mathcal{V}$ while $\mathcal{X}_{-1} = \{p_k\}$ is the basis of polynomial functions $\mathcal{V} \to K$. Then the matrix $T_{\mathcal{X}F}$ is block diagonal with the same block along the diagonal*

$$T_{\mathcal{X}_{-1}F_{-1}} = \sum_{k,j} p_k \otimes e_n^{\partial}(\tilde{f}_j). \tag{52}$$

Corollary 30 *When the tensor $T_{\mathcal{X}F}$ (48) of basis transformation $F \to \mathcal{X}$ is block diagonal as by Theorem 29, the tensor $T_{F\mathcal{X}} = T_{\mathcal{X}F}^{-1}$ (47) of basis transformation $\mathcal{X} \to F$ is found by simply inverting each block (52).*

Note however, that this special case can not model basic operations that change several memory locations at once, while the general model presented in this paper can. Also note that the main goal of this work is to develop methods for analysis of computer programs, making the programming spaces of maps $\mathcal{V} \to \mathcal{V}$ much more appropriate than the programming spaces of functions.

5.8 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 (as a spline).

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 $\vec{x} \in \mathcal{V}$

$$P(\vec{x}) = \begin{cases} P_{n_1 1} \circ P_{(n_1-1)1} \circ \dots P_{11}(\vec{x}); & \vec{x} \in \Omega_1 \\ P_{n_2 2} \circ P_{(n_2-1)2} \circ \dots P_{12}(\vec{x}); & \vec{x} \in \Omega_2 \\ \vdots & \vdots \\ P_{n_k k} \circ P_{(n_k-1)k} \circ \dots P_{1k}(\vec{x}); & \vec{x} \in \Omega_k \end{cases} \tag{53}$$

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

$$e^{\partial}P(\vec{x}) = \begin{cases} e^{\partial}P_{n_1 1} \circ e^{\partial}P_{(n_1-1)1} \circ \dots \circ e^{\partial}P_{11}(\vec{x}); & \vec{x} \in \mathrm{int}(\Omega_1) \\ e^{\partial}P_{n_2 2} \circ e^{\partial}P_{(n_2-1)2} \circ \dots \circ e^{\partial}P_{12}(\vec{x}); & \vec{x} \in \mathrm{int}(\Omega_2) \\ \vdots & \vdots \\ e^{\partial}P_{n_k k} \circ e^{\partial}P_{(n_k-1)k} \circ \dots \circ e^{\partial}P_{1k}(\vec{x}); & \vec{x} \in \mathrm{int}(\Omega_k) \end{cases} \tag{54}$$

Theorem 31 *Each program $P \in \mathcal{P}$ containing control structures is infinitely-differentiable on the domain $\Omega = \bigcup_{\forall_i} \mathrm{int}(\Omega_i)$.*

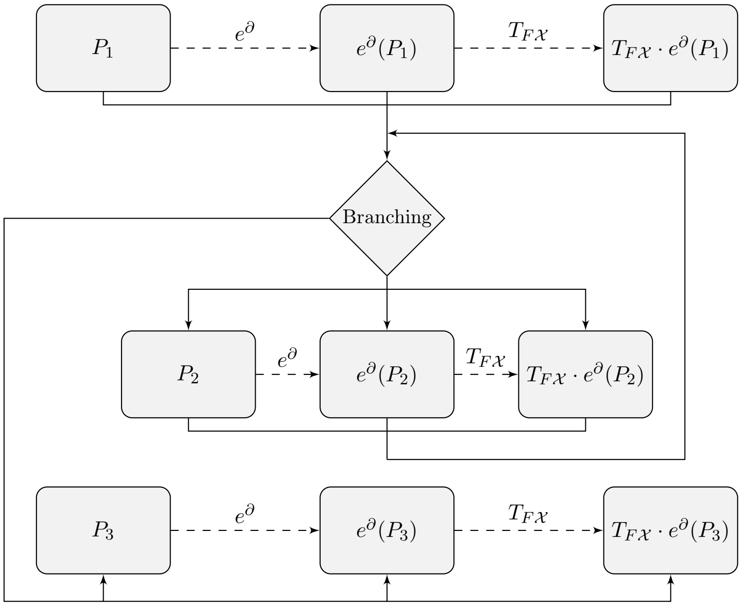

Figure 1: Transformation diagram

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 9 through argument (20). ■

Branching of programs into domains (53) is done through conditional statements. Each conditional causes a branching in programs' execution tree.

Proposition 32 *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.*

Remark 33 *Iterators, that do not change exit conditions within its' body, do not cause branching.*

This section concerns itself with employing the derived theorems to propose a linear treatment of branchings and avoid the exponential threat of Proposition 32 to applications of the theory.

Theorem 34 *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 21, 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 21.

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 9, each of these branches is analytic. ■

Remark 35 *Images of the operator e^{∂} and $T_{F\mathcal{X}}$ are elements of the original space $\mathcal{P}$, which may be composed. Thus for $P = p_3 \circ p_2 \circ p_1$, the following makes sense*

$$P = \Big(p_3 \circ e^{\partial}(p_2) \circ T_{F\mathcal{X}} e^{\partial}(p_1)\Big) \in \mathcal{P}$$

The same holds true for all permutations of applications of operators e^{∂}, $T_{F\mathcal{X}}$ and id, as visible in Figure 1.

Remark 36 *In practice, we always use projections of the operator e^{∂} to some finite order n, resulting in e^{∂}_n. Therefore, we must take note that the following relation holds*

$$e^{\partial}_m(P_2) \circ e^{\partial}_n(P_1) = e^{\partial}_k(P_2 \circ P_1) \iff 0 \leq k \leq \min(m,n)$$

when composing two images of the applied operator, projected to different subspaces.

The transformation tensor $T_{F\mathcal{X}}$ is needed to be computed only once and can then on be applied to any program running on said hardware. The same holds true for each branch p_i, which can, by Theorem 24, be freely composed amongst each other.

Remark 37 *Images of the operator $e^{\partial}(P \in \mathcal{P}_0)$ are elements of $\mathcal{V} \otimes T(\mathcal{V}^*)$ by (28), consisting of multi-linear maps. As such, their evaluation and composition $(e^{\partial}(P_1) \circ e^{\partial}(P_2))$ is tailor made for efficient implementation by methods of parallelism (e.g., Abdelfattah and et al., 2016), with computable complexities.*

6. Generalized Tensor Networks

Let $\mathbf{W} = \sum\limits_{i=0}^{n}(\mathbf{w}_i \in \mathcal{V} \otimes \mathcal{V}^{*\otimes i})$ be an element of the virtual memory $\mathcal{V} \otimes T_n(\mathcal{V}^*)$ in an analytic machine of Definition 15. The element $\mathbf{W}$ can be seen as a map $\mathbf{v} \mapsto \mathbf{W}(\mathbf{v})$ as defined by (31):

$$\mathbf{W}(\mathbf{v}) = \sum_{i=0}^{n} \mathbf{w}_i \cdot \mathbf{v}^{\otimes i} \in \mathcal{V}.$$

Definition 38 *A* general tensor network $\mathcal{N}$ *is a sequence of maps $L_i : \mathcal{V} \to \mathcal{V}$ called* layers, *defined recursively by the equation*

$$L_0 = \text{id}; \quad L_{i+1} = \Phi_i \circ \mathbf{W}_i \circ L_i$$

where $\mathbf{W}_i \in \bigoplus_{k=0}^{n} \mathcal{V} \otimes \mathcal{V}^{*k\otimes}$ *are the* weights *(and the bias) and* $\Phi_i \in \mathcal{P}_0$ *is the* activation function.

We may look at a general tensor network with n layers as a map. In that case we mean

$$\mathcal{N} = L_n \in \mathcal{P}_0.$$

Remark 39 *The common neural network is a tensor network with* $\mathbf{b}_i + \mathbf{W}_i \in \mathcal{V} \oplus \mathcal{V} \otimes \mathcal{V}^*$ *as weight multi-tensors.*

Generalizations of recurrent (Socher et al., 2011), convolutional (Krizhevsky et al., 2012) and deep residual (He et al., 2015) neural networks, and mechanisms such as long short term memory (Hochreiter and Schmidhuber, 1997), are easily generated by this model, as they are all elements of a differential programming space $\mathcal{P}_0$.

A general tensor network $\mathcal{N}$ can be represented by a deeper common neural network. This equivalence means, that a general tensor network with fewer layers can provide the same results as a deeper common neural network, while mitigating the vanishing gradient problem described by Pascanu et al. (2013) that occurs in training due to the depth of the network coupled with machine precision.

Remark 40 *Existing architectures like Theano by Theano Development Team (2016), TensorFlow by Abadi and et al. (2015) and others could be easily adapted to handle general tensor networks.*

6.1 Programs as General Tensor Networks

Let $P = P_n \circ \cdots \circ P_0 \in \mathcal{P}_0$ be the procedure of interest, with $P_i \in \mathcal{P}_0$ being the source code between two branching points, like shown in Figure 1. By Theorem 17 we have

$$P(\mathbf{v}_0 + \mathbf{v} \in \mathcal{V}) = e^{\partial}|_{\mathbf{v}_0} P(\mathbf{v} \in \mathcal{V}) \tag{55}$$

and through Corollary 24

$$P(\mathbf{v}_0 + \mathbf{v} \in \mathcal{V}) = e^{\partial}|_{\mathbf{v}_n} P_n \circ \cdots \circ e^{\partial}|_{\mathbf{v}_0} P_0(\mathbf{v} \in \mathcal{V}) \tag{56}$$

which is the transformation hereon denoted by $e^{\partial} P$.

Proposition 41 *The image of the application of the operator* e^{∂} *to a program* $P \in \mathcal{P}_0$ *as in* (56)*, is a general tensor network, with the activation function* Φ_i *being the identity map, at each layer.*

6.1.1 Transformations of Programs to General Tensor Networks

The transformed program (55) equals the original program by Theorem 17 and Corollary 24. But in practice, we are always working with a finite virtual memory $\mathcal{V} \otimes T_n(\mathcal{V}^*)$ and the equality becomes an approximation. Thus we treat the transformation of the original program, as the initialization of the weights (and the bias) of a general tensor network to be trained. This motivates modernizing the historic Taylor series and evolving the *generalized shift operator* (18).

Definition 42 (Neural tensor series) *Assume the program $P \in \mathcal{P}_0$ can be written as a composition $P = P_n \circ \cdots \circ P_0 \in \mathcal{P}_0$. A general tensor network $\mathcal{N}_\Phi|_{\mathbf{v}_0}P$ defined with a set of activation functions $\Phi = \{\Phi_k \in \mathcal{P}_0 : \mathcal{V} \to \mathcal{V}; \quad 0 \leq k \leq n\}$ and weights*

$$\mathbf{W}_k = e_N^\partial|_{\mathbf{v}_k}P_k; \quad \mathbf{v}_k = P_k \circ \cdots P_0(\mathbf{v}_0)$$

is called neural tensor series of order N *for a program* P *at a point* $\mathbf{v}_0$ *with activation functions* Φ *with final layer*

$$\mathcal{N}_\Phi|_{\mathbf{v}_0}P = \Phi_n \circ e_N^\partial|_{\mathbf{v}_n}P_n \circ \cdots \circ \Phi_0 \circ e_N^\partial|_{\mathbf{v}_0}P_0. \tag{57}$$

Remark 43 *Neural tensor series transforms a common program to a trainable general tensor network, naturally extending Theorem 34 and the Transformation diagram of Figure 1. For example, by setting $N = 1$ in* (57) *a program can be transformed to a common neural network.*

Definition 42 has wide applications to different fields. In practice if a sub-optimal algorithm providing an approximate solution is available, the *neural tensor series* serves as a great initialization point for further training, leading to a process of *boosting* (Freund and et al., 1999) converting a weaker learner to a strong one. As currently neural networks give best results to many problems, the described method is likely to provide improvements to existing methods.

As each general tensor network is a neural tensor series of some program, it provides an elegant way of expressing neural computations through operational calculus. This has direct implications to the study and understanding of concepts such as *differentiable neural computers* (Graves and et al., 2016) and *neural programmer-interpreters* (Reed and de Freitas, 2016). Interchanging neural processes and programming spaces reveals new directions to explore, as we might gain insight on one by identifying it with the other. By this equivalence, the relation between a computer program and a general tensor network is that of a function and its Taylor series enhanced by activations. Thus, they may be appropriately employed to the purpose of analysis in computer science, a path well walked by other fields.

Remark 44 *All coefficients $\mathbf{W}_i \in \mathcal{V} \otimes T(\mathcal{V}^*)$ are multi-linear maps allowing efficient implementation through GPU parallelism as by Claim 37. Thus, general tensor networks may employ it, just as the common neural networks they generalize do.*

6.1.2 Compositions of Tensor Networks with General Programs

Methods for control structures and branching presented in Section 5.8 apply to general tensor networks.

Proposition 45 *An arbitrary $P \in \mathcal{P}_0$ can be composed with general tensor networks $\mathcal{N}_i$*

$$\tilde{P} = \mathcal{N}_2 \circ P \circ \mathcal{N}_1 \in \mathcal{P}_\infty \tag{58}$$

and $\tilde{P}$ is an element of a differentiable programming space $\mathcal{P}_\infty$, and can thus be treated by the operational calculus.

Remark 46 *Any layer $L_i \in \mathcal{P}_n$ can be composed with an arbitrary element of the differentiable programming space $\mathcal{P}_0$. This allows algorithmic coding of trainable memory managers, generalizing concepts such as long short term memory (Hochreiter and Schmidhuber, 1997) and easing the implementation of networks capable of reading from and writing to an external memory (Graves and et al., 2016), by freeing semantics of their design process.*

6.2 Training of General Tensor Networks

All transformed programs $\mathcal{N}_\Phi|_{\mathbf{v}_0}P$ are elements of a differentiable programming space $\mathcal{P}_\infty$. As such, the operational calculus, and the operators it offers can freely be applied to them.

By Corollary 24 we have

$$e_n^\partial(L_{i+1}) = e_n^\partial(\Phi_i) \circ e_n^\partial(W_i \circ L_i) \in \mathcal{P}_n \tag{59}$$

Thus by Corollary 27, the n-differentiable k-th derivatives can be extracted by

$${}^{n}L_{i+1}^{k\prime} = \phi^k \circ e_{n+k}^\partial(L_{i+1}) \in \mathcal{P}_n$$

from (59), where ϕ is the reduction of order map of Theorem 26. Derivatives of specific order are extracted by projecting on components of P_n, and can be used in any of the well established training methods in the industry.

Proposition 47 *Using the operator* $\exp(\partial_f e^{h\partial_g})$ *of Theorem 21, both forward and reverse mode automatic differentiation (generalized to arbitrary order) can be implemented on general tensor networks.*

Remark 48 *The operational calculus can be applied to the training process $T \in \mathcal{P}_0$ of a general tensor network itself, as it is an element of a differentiable programming space $\mathcal{P}_0$. Thus, hyper-parameters of the training process can be studied (e.g., Bengio, 2000), analyzed and trained (e.g., Thornton and et al., 2013), as to be adapted to the particulars of the problem being solved.*

By Proposition 45, any training methods enabled by the operational calculus presented in this paper apply to all compositions of general tensor networks $\mathcal{N}_i$ with arbitrary programs $P \in \mathcal{P}$. This allows seamless trainable transitions between code formulations, naturally extending the Transformation diagram of Figure 1.

7. Analysis

Operational calculus offers new approaches to program analysis that are the object of study in this section. We demonstrate how to intertwine algorithmic control flow, operational calculus and algebra.

7.1 Study of Properties

We will denote the fact, that some object $\mathbf{v}$ has the property X, by $\mathbf{v} \in X$. Suppose we have $\mathbf{v} \notin X$, and desire a procedure $P \in \mathcal{P}_0$, that in some way modifies $\mathbf{v}$ to have the property X, changing the element $\mathbf{v}$ as little as possible:

$$P \in \mathcal{P}_0 : \mathbf{v} \notin X \to P(\mathbf{v}) \in X \tag{60}$$

Usually such procedures are difficult to construct and may not even explicitly exist. An easier task is to construct a procedure $T \in \mathcal{P}_0$, whose output allows deduction of whether $\mathbf{v}$ has the property X or not.

We propose an algorithm

$$A : T \in \mathcal{P} \to P \in \mathcal{P} \tag{61}$$

transforming a procedure T testing for a property X_j, to a procedure $P \in \mathcal{P}_0 : \mathbf{v} \notin X_j \to P(\mathbf{v}) \in X_j$ imposing that property onto any object in the domain $\Omega \subset \mathcal{V}$.

We employ operational calculus as we probe procedures' inner structure and explore how it interacts with the elements of the domain.

7.1.1 Activity Profiles and Property Measures

To be able to determine which conceptual steps are the most important for the result of the testing procedure T, we have to somehow measure the activity at that step. A simple example of the measure of activity $\mathcal{A}$ could simply be the norm of the appropriate derivative, as it measures the rate of change.

Let's assume

$$T = T_n \circ T_{n-1} \circ \cdots \circ T_1$$

for simplicity[9]. The part T_i is called a conceptual step in the procedure.

Definition 49 (Activity profile) *The function $\mathcal{A}_i : \Omega \to [0,1]$ is called a* measure of activity *in the conceptual step T_i. The value*

$$\mathcal{A}_i \circ e^{\partial_{T_i}} T(\mathbf{v})$$

is the activity level *of T_i for a given element of the domain $\mathbf{v} \in \Omega$ taken as the input of T*[10]*.*

The vector

$$\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n) : \Omega \to [0,1]^n$$

represents the activity profile of the procedure T.

Definition 50 (Property measure) *A function*

$$M_X \in \mathcal{P}_0 : [0,1]^n \to [0,1]$$

is called the property measure, measuring the amount of property X in an object $v \in \Omega$, if there exists $c \in (0,1)$ such that

$$\mathbf{v} \in X \iff M_x \circ \mathcal{A}(\mathbf{v}) \geq c.$$

[9] The procedure T could also be piecewise-differentiable as in (53), by specific T_i containing a control structure as is ilustrated in Figure 1

[10] The operator $e^{\partial_{T_i}}$ considers only the variable parameters of the sub-procedure T_i as variables

Algorithm 1 Construct property measure

```
procedure Construct property measure
    for each $\mathbf{v}_k \in \Omega$ do
        for each $T_i$ do
            extract $a_k^i = \mathcal{A}_i \circ e^{\partial_{T_i}} T(\mathbf{v}_k)$
        end for
    end for
    initialize set $I$
    for each $X_j$ do
        generate property measure $M_{X_j}$ from $a_k^i$
        add $M_{X_j}$ to $I$
    end for
    return $I$
end procedure
```

Once the activity profiles and property measures are generated, starting with an element, without the property X_j, we can use any established optimization technique, to optimize and increase the measure M_{X_j}. When the increase of the measure M_{X_j} is sufficient, this results in an new object possessing the property X_j.

Theorem 51 *With an appropriate choice of the activity profile $\mathcal{A}$, the Algorithm 2 transforms a procedure $T \in \mathcal{P}$, testing for a property X_j, to a procedure P, that increases the property measure of any object in the domain.*

Corollary 52 *By Theorem 51, existence of a procedure testing an object for validity is sufficient for constructing a procedure transforming a non-valid object to a valid one*

$$P \in \mathcal{P}_0 : \mathbf{v} \notin X_j \to P(\mathbf{v}) \in X_j$$

under the assumption that the increase of the property measure is sufficient.

When T serves as a simulation, with $\mathbf{v}$ modeling a real-life object of study, the procedure opens new doors in analyzing real-life phenomena. We may observe how $\mathbf{v}$ evolves through iterations and study stages of change, which serves as a useful insight when designing procedures causing change in real-life phenomena.

Algorithm 2 Appoint property X_j to $\mathbf{v} \in \Omega$

```
procedure Appoint property $X_j$ to $\mathbf{v} \in \Omega$
    initialize path $\gamma$ with $\mathbf{v}$
    for each step do
        for each $T_i \in T_{X_j}$ do
            extract $T_i' = \phi \circ e_2^{\partial_{\mathbf{v}}}(T_i \circ \cdots \circ T_1) \in \mathcal{P}_1$
            compute the energy $E_i = e_1^{\partial_{\mathbf{v}}}(M_{X_j}) \circ T_i' \in \mathcal{P}_1$
            extract the derivative $\partial_v E_i = \mathrm{pr}_{\{\partial\}}(E_i)$
            add $\partial_{\mathbf{v}} E_i$ to $\partial_{\mathbf{v}} E$
        end for
        update $\mathbf{v}$ by $step(\partial_{\mathbf{v}} E, \mathbf{v})$
        insert $\mathbf{v}$ to $\gamma$
    end for
    return $\gamma$
end procedure
```

7.1.2 Example

The derived procedures P given by Algorithm 2 are generalizations of methods already present in practice. This example demonstrates how the method can be employed.

We take the measure of activity to be the norm of the derivative with respect to the variable parameters in sub-procedure T_i.

$$\|\partial_{T_i} T\| = \left\| \mathrm{pr}_{\{\partial\}} \left(e_1^{\partial_{T_i}}(T) \right) \right\|$$

Then for each property X_j we select the set of sub-procedures

$$T_{X_j} = \{ T_i \in T_{X_j} \iff \|\partial_{T_i} T\| \geq c \}$$

that have the highest measure of activity at the elements of X_j. The property measure for the property X_j is then simply the sum of squares of norms of the derivatives of selected sub-procedures $T_1^i = T_i \circ \cdots \circ T_1$. This completes Algorithm 1.

By Corollary 27, n-differentiable k-th derivatives ${}^nT_i^{k\prime} \in \mathcal{P}_n$ with respect to the input $\mathbf{v} \in \Omega$ are computed by

$${}^nT_i^{k\prime} = \phi^k \circ e_{n+k}^{\partial_{\mathbf{v}}}(T_1^i) \in \mathcal{P}_n$$

where ϕ is the reduction of order map of Theorem 26. Using $\|\cdot\|^2 \in \mathcal{P}_1$ as the norm map, the property measure is

$$M_{X_j} = \sum_{T_i \in T_{X_j}} \|T_i'\|^2 \in \mathcal{P}_1$$

assuming it only needs to be once differentiable. Optimization of the property measure completes Algorithm 2.

When $T \in \mathcal{P}_0$ represents a neural network, T_i stands for a specific layer in the network, with neurons being its variable parameters, the Algorithm 2 gives a procedure, that acts similarly to Google's Deep Dream Project (see Mordvintsev et al., 2015) and Neural Algorithm of Artistic Style by Gatys et al. (2015), as they are special cases of Algorithm 2. However Algorithm 2 may be applied to any program $T \in \mathcal{P}_0$, not just neural networks.

8. Conclusions

Existence of a program is embedded in a virtual reality, forming a system of objects undergoing change in a virtual space. Just as the reality inhabited by us is being studied by science, revealing principles and laws, so can the virtual reality inhabited by programs. Yet here lies a tougher task, as the laws of the system are simultaneously observed and constructed; the universe is bug-free, up to philosophic precision, while our programs are not. This reinforces the need for a language capable of not only capturing, but also constructing digital phenomena, a feat demonstrated by analytic virtual machines and operational calculus.

Inspired by the endeavors of Feynman (1951) and Heaviside (see Carson, 1922) in physics before us, we applied operational calculus to programming spaces, allowing analytic conclusions through algebraic means, easing implementation. It yielded the operator of program composition, generalizing both forward and reverse mode of automatic differentiation to arbitrary order, under a single operator in the theory. Both the use of algebra and operational calculus were demonstrated, as calculations and manipulations were performed on the operator level, before the operator is applied to a particular program. The language presented in this work condenses complex notions into simple expressions, enabling formulation of meaningful algebraic equations to be solved. In doing so, functional transformations of programs in arbitrary function basis' were derived, a useful tool when adapting code to the specifics of a given hardware, among other. All such formulations are invariant not only to the choice of a programming space, but also to the point in execution of a program, introducing the principle of general covariance to programming. Offerings of this principle were exploited in designing methods on how transformations are to be interchangeably applied in practice in Section 5.8. These methods allow seamless transitions between transformed forms and original code throughout the program.

Operational calculus provides more than mere means to calculating derivatives. Its depth allows merging modern discoveries with known old truths in the form of Neural Tensor series. A construct most useful when coming to terms with the finite horizon beyond which theoretical truths become pragmatic approximations. It allows us to treat the idealistic case deprecated by the finite, as an initial prediction of a model to be trained. This has widespread applicability, as it can be seen as a process of *boosting*, which already enriched other fields of science.

As each general tensor network is a Neural tensor series of some program, we might gain insight on one by identifying it with the other, hopefully further bridging the gap in understanding between continuous and discrete computation. Operational calculus provides a rigorous framework for such discussion, as was briefly explored in the final section, and will be the authors' next subject of study.

Acknowledgments

First author extends his gratitude to dr. Jure Kališnik for proofreading the proofs, and prof. Borut Robič for general guidance through the academia.

References

Martín Abadi and et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

A. Abdelfattah and et al. High-performance tensor contractions for {GPUs}. *Procedia Computer Science*, 80:108 – 118, 2016. ISSN 1877-0509. International Conference on Computational Science 2016, {ICCS} 2016, 6-8 June 2016, San Diego, California, {USA}.

Atilim Gunes Baydin and et. al. *Automatic differentiation in machine learning: a survey.* 2015.

Yoshua Bengio. Gradient-based optimization of hyperparameters. *Neural computation*, 12 (8):1889–1900, 2000.

John R. Carson. The heaviside operational calculus. *Bell System Technical Journal*, 1(2): 43–55, 1922. ISSN 1538-7305.

Richard P. Feynman. An operator calculus having applications in quantum electrodynamics. *Phys. Rev.*, 84:108–128, Oct 1951.

Yoav Freund and et al. A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence*, 14(771-780):1612, 1999.

Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style. *arXiv e-prints*, arXiv:1508.06576, 2015.

Mark Girolami and Ben Calderhead. Riemann manifold Langevin and Hamiltonian Monte Carlo methods. *Journal of the Royal Statistical Society*, 2011.

Alex Graves and et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, advance online publication, Oct 2016. ISSN 1476-4687. Article.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv e-prints*, 2015.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Robin J. Hogan. Fast reverse-mode automatic differentiation using expression templates in c++. *ACM Trans. Math. Softw.*, 40(4):26:1–26:16, July 2014. ISSN 0098-3500.

Kamil A. Khan and Paul I. Barton. A vector forward mode of automatic differentiation for generalized derivative evaluation. *Optimization Methods and Software*, 30(6):1185–1212, 2015.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Alexander Mordvintsev, Christopher Olah, and Mike Tyka. Inceptionism: Going deeper into neural networks, 2015. URL `https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html`.

O'Hanian, Hans C., Ruffini, and Remo. *Gravitation and Spacetime (2nd ed.)*. W. W. Norton, 1994. ISBN 0-393-96501-5.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *ICML (3)*, 28:1310–1318, 2013.

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

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

Scott Reed and Nando de Freitas. Neural programmer-interpreters. In *International Conference on Learning Representations (ICLR)*, 2016. URL `http://arxiv.org/pdf/1511.06279v3`.

Richard Socher, Cliff C Lin, Chris Manning, and Andrew Y Ng. Parsing natural scenes and natural language with recursive neural networks, 2011.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, arXiv:1605.02688, May 2016.

Chris Thornton and et al. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855. ACM, 2013.

Norbert Wiener. The operational calculus. *Mathematische Annalen*, 95:557–584, 1926.

Žiga Sajovic. dcpp, 2016a. URL `https://github.com/zigasajovic/dCpp`.

Žiga Sajovic. Implementing operational calculus on programming spaces for differentiable computing. *arXiv e-prints*, arXiv:1612.0273, 2016b.