Backpropagation in the Simply Typed Lambda-Calculus with Linear Negation

ALOÏS BRUNEL, Deepomatic, France
DAMIANO MAZZA, CNRS, France
MICHELE PAGANI, Université de Paris, France

Backpropagation is a classic automatic differentiation algorithm computing the gradient of functions specified by a certain class of simple, first-order programs, called computational graphs. It is a fundamental tool in several fields, most notably machine learning, where it is the key for efficiently training (deep) neural networks. Recent years have witnessed the quick growth of a research field called differentiable programming, the aim of which is to express computational graphs more synthetically and modularly by resorting to actual programming languages endowed with control flow operators and higher-order combinators, such as map and fold. In this paper, we extend the backpropagation algorithm to a paradigmatic example of such a programming language: we define a compositional program transformation from the simply-typed lambda-calculus to itself augmented with a notion of linear negation, and prove that this computes the gradient of the source program with the same efficiency as first-order backpropagation. The transformation is completely effect-free and thus provides a purely logical understanding of the dynamics of backpropagation.

CCS Concepts: • **Theory of computation → Program semantics**; *Theory and algorithms for application domains.*

Additional Key Words and Phrases: Differentiable Programming, Lambda-Calculus, Linear Logic

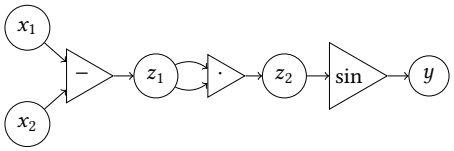
ACM Reference Format:
Aloïs Brunel, Damiano Mazza, and Michele Pagani. 2020. Backpropagation in the Simply Typed Lambda-Calculus with Linear Negation. *Proc. ACM Program. Lang.* 4, POPL, Article 64 (January 2020), 27 pages. https://doi.org/10.1145/3371132

1 INTRODUCTION

In the past decade there has been a surge of interest in so-called *deep learning*, a class of machine learning methods based on multi-layer neural networks. The term "deep" has no formal meaning, it is essentially a synonym of "multi-layer", which refers to the fact that the networks have, together with their input and output layers, at least one internal (or "hidden") layer of artificial neurons. These are the basic building blocks of neural networks: technically, they are just functions $\mathbb{R}^m \to \mathbb{R}$ of the form $(x_1, \ldots, x_m) \to \sigma\left(\sum_{i=1}^m w_i \cdot x_i\right)$, where $\sigma : \mathbb{R} \to \mathbb{R}$ is some *activation function* and $w_1, \ldots, w_m \in \mathbb{R}$ are the *weights* of the neuron. A layer consists of several unconnected artificial neurons, in parallel. The simplest way of arranging layers is to connect them in cascade, every output of each layer feeding into every neuron of the next layer, forming a so-called *feed-forward architecture*. Feed-forward, multi-layer neural networks are known to be universal approximators: any continuous function $f : K \to \mathbb{R}$ with $K \subseteq \mathbb{R}^k$ compact may be approximated to an arbitrary

Authors' addresses: Aloïs Brunel, Deepomatic, France, alois.brunel@gmail.com; Damiano Mazza, CNRS, UMR 7030, LIPN, Université Sorbonne Paris Nord, France, Damiano.Mazza@lipn.univ-paris13.fr; Michele Pagani, IRIF UMR 8243, Université de Paris, CNRS, France, pagani@irif.fr.

2475-1421/2020/1-ART64
https://doi.org/10.1145/3371132




```
let z1 = x1 - x2 in let z2 = z1 · z1 in sin z2
```

Fig. 1. A computational graph with inputs x_1, x_2 and output y, and its corresponding term. Nodes are drawn as circles, hyperedges as triangles. The output y does not appear in the term: it corresponds to its root.

degree of precision by a feed-forward neural network with one hidden layer, as long as the weights are set properly [Cybenko 1989; Hornik 1991]. This leads us to the question of *how to efficiently train a neural network*, *i.e.*, how to find the right weights as quickly as possible.

Albeit in theory the simplest architecture suffices for all purposes, in practice more complex architectures may perform better, and different architectures may be better suited for different purposes. We thus get to another basic question of deep learning: *how to select and, if necessary, modify or design network architectures adapted to a given task*. Since the quality of an architecture is also judged in terms of training efficiency, this problem is actually interlaced with the previous one.

The first question is generally answered in terms of the *gradient descent* algorithm (or some variant thereof). This finds local minima of a function $G : \mathbb{R}^n \to \mathbb{R}$ using its gradient ∇G, *i.e.*, the vector of the partial derivatives of G (Equation 1). The algorithm starts by choosing a point $\mathbf{w}_0 \in \mathbb{R}^n$. Under certain assumptions, if $\nabla G(\mathbf{w}_0)$ is close to zero then $\mathbf{w}_0$ is within a sufficiently small neighborhood of a local minimum. Otherwise, we know that G decreases most sharply in the opposite direction of $\nabla G(\mathbf{w}_0)$, and so the algorithm sets $\mathbf{w}_1 := \mathbf{w}_0 - \rho \nabla G(\mathbf{w}_0)$ for a suitable step rate $\rho > 0$, and repeats the procedure from $\mathbf{w}_1$. In the case of deep learning, a neural network (with one output) induces a function $h(\mathbf{w}, \mathbf{x}) : \mathbb{R}^{n+k} \to \mathbb{R}$, where k is the number of inputs and n the number of weights of all the neurons in the network. By fixing $\mathbf{w} \in \mathbb{R}^n$ and making $\mathbf{x}$ vary over a set of *training inputs*, we may measure how much h differs from the target function $f : \mathbb{R}^k \to \mathbb{R}$ when the weights are set to $\mathbf{w}$. This defines an error function $G : \mathbb{R}^n \to \mathbb{R}$, the minimization of which gives a way of finding the desired value for the weights. The training process thus becomes the iteration, over and over, of a gradient computation, starting from some initial set of weights $\mathbf{w}_0$.

So, regardless of the architecture, efficiently training a neural network involves efficiently computing gradients. The interest of gradient descent, however, goes well beyond deep learning, into fields such as physical modeling and engineering design optimization, each with numerous applications. It is thus no wonder that a whole research field, known as *automatic differentiation* (AD for short), developed around the computation of gradients and, more generally, Jacobians[1] [Baydin et al. 2017]. In AD, the setting of neural networks is generalized to *computational graphs*, which are arbitrarily complex compositions of nodes computing basic functions and in which the output of a node may be shared as the input of an unbounded number of nodes. Fig. 1 gives a pictorial representation of a simple computational graph made of only three basic functions (subtraction, multiplication and sine), with inputs x_1 and x_2. Notice that the computation of $x_1 - x_2$ is shared by the two factors of the multiplication. Neural networks are special cases of computational graphs.

The key idea of AD is to compute the gradient of a computational graph by accumulating in a suitable way the partial derivatives of the basic functions composing the graph. This rests on the mathematical principle known as *chain rule*, giving the derivative of a composition $f \circ g$ from the derivatives of its components f and g (Equation 3). There are two main "modes" of applying this rule

[1]The generalization of the gradient to the case of functions $\mathbb{R}^n \to \mathbb{R}^m$ with $m > 1$.

in AD, either *forward*, propagating derivatives from inputs to outputs, or *backwards*, propagating derivatives from outputs to inputs. As will be explained in Sect. 2, if G is a computational graph with n inputs and m outputs invoking $|G|$ operations (*i.e.*, nodes), forward mode computes the Jacobian of G in $O(n\,|G|)$ operations, while reverse mode requires $O(m\,|G|)$ operations. In deep learning, as the number of layers increases, n becomes astronomical (millions, or even billions) while $m = 1$, hence the reverse mode is the method of choice and specializes in what is called the *backpropagation algorithm* (Sect. 2.4), which has been a staple of deep learning for decades [LeCun et al. 1989]. Today, AD techniques are routinely used in the industry through deep learning frameworks such as TensorFlow [Abadi et al. 2016] and PyTorch [Paszke et al. 2017].

The interest of the programming languages (PL) community in AD stems from the second deep learning question mentioned above, namely the design and manipulation of (complex) neural network architectures. As it turns out, these are being expressed more and more commonly in terms of actual programs, with branching, recursive calls or even higher-order primitives, like list combinators such as `map` or `fold`, to the point of yielding what some call a generalization of deep learning, branded *differentiable programming* [LeCun 2018]. Although these programs always reduce, in the end, to computational graphs, these latter are inherently static and therefore inadequate to properly describe something which is, in reality, a dynamically-generated neural network. Similarly, traditional AD falls short of providing a fully general foundation for differentiable programming because, in order to compute the gradient of an *a priori* arbitrary (higher order) program, it forces us to reduce it to a computational graph first. In PL-theoretic terms, this amounts to fixing a reduction strategy, which cannot always be optimal in terms of efficiency. There is also a question of modularity: if gradients may only be computed by running programs, then we are implicitly rejecting the possibility of computing gradients modularly, because a minor change in the code might result in having to recompute everything from scratch, which is clearly unsatisfactory.

This paper is a contribution to the theoretical foundations of differentiable programming. We define a compositional program transformation $\overleftarrow{\mathbf{D}}$ (Fig. 5) extending the backpropagation algorithm from computational graphs to general simply typed λ-terms. Our framework is purely logical and therefore offers the possibility of importing tools from semantics, type systems and rewriting theory. The benefit is at least twofold, in the form of

(1) a soundness proof (Theorem 15 and Corollary 16), which relies on the logical/compositional definition of $\overleftarrow{\mathbf{D}}$ and the semantics;
(2) a complexity analysis, which hinges on rewriting in the form of the *linear substitution calculus* [Accattoli 2012] and which guarantees that generalized backpropagation is at least as efficient as the standard algorithm on computational graphs.

Although the soundness proof is based on a fixed strategy (reducing to a computational graph first and then computing the gradient), the confluence of our calculus guarantees us that the gradient may be computed using any other reduction strategy, thus allowing arbitrary evaluation mechanisms for executing backpropagation in a purely functional setting, without necessarily passing through computational graphs. Sect. 6 discusses the benefits of this in terms of efficiency.

On a similar note, compositionality ensures modularity: for instance, if $p = tu$, *i.e.*, program p is composed of subprograms t and u, then $\overleftarrow{\mathbf{D}}(p) = \overleftarrow{\mathbf{D}}(t)\overleftarrow{\mathbf{D}}(u)$ and $\overleftarrow{\mathbf{D}}(t)$ and $\overleftarrow{\mathbf{D}}(u)$ may be computed independently, so that if p is modified into tu', the computation of $\overleftarrow{\mathbf{D}}(t)$ may be reused. Modularity also opens the way to parallelization, another potential avenue to efficiency.

Finally, let us stress that the transformation $\overleftarrow{\mathbf{D}}$ is remarkably simple: on "pure" λ-terms, *i.e.*, not containing any function symbol corresponding to the basic nodes of computational graphs, $\overleftarrow{\mathbf{D}}$ is the identity, modulo a change in the types. In particular, $\overleftarrow{\mathbf{D}}$ maps a datatype constructor/destructor

(`cons`, `head`, `tail`, etc.) or a typical higher order combinator (`map`, `fold`, etc.) to itself, which makes the transformation particularly easy to compute and analyze. We see this as a theoretical counterpart to the use of *operator overloading* in implementations of AD.

From a more abstract perspective, all these properties (including compositionality) may be succinctly summarized in the fact that $\overleftarrow{\mathbf{D}}$ is a *cartesian closed 2-functor* or, better, a *morphism of cartesian 2-multicategories*, obtained by *freely lifting to λ-terms a morphism defined on computational graphs*. However, such a viewpoint will not be developed in this paper.

Related work. Our main source of inspiration is [Wang et al. 2019], where a program transformation $\overleftarrow{\mathcal{D}}$ is proposed as a compositional extension of symbolic backpropagation to functional programs. We summarize their approach in Sect. 3, let us concentrate on the main differences here:

(1) the transformation $\overleftarrow{\mathcal{D}}$ uses references and delimited continuations, while our transformation $\overleftarrow{\mathbf{D}}$ is purely functional and only relies on a linear negation primitive on the ground type. Albeit [Wang et al. 2019] do mention that a purely functional version of their transformation may be obtained by encoding the memory inside the type of the continuation, this encoding adds a sequentialization (introduced by the order of the memory updates) which is absent in $\overleftarrow{\mathbf{D}}$ and which makes our transformation more amenable to parallelization.

(2) The transformation $\overleftarrow{\mathcal{D}}$ applies to a Turing-complete programming language, while we focus on the much more restrictive simply-typed λ-calculus. However, no soundness proof for $\overleftarrow{\mathcal{D}}$ is given, only testing on examples. This brings to light a difference in the general spirit of the two approaches: the paper [Wang et al. 2019] is mainly experimental, it comes with a deep learning framework (Lantern) and with a case study showing the relevance of this line of research. Our approach is complementary because it is mainly theoretical: we focus on proving the soundness and complexity bound of a non-trivial differentiable programming framework, "non-trivial" in the sense that it is the simplest exhibiting the difficulty of the task at hand. To the best of our knowledge, such foundational questions have been completely neglected, even for "toy" languages, and our paper is the first to address them.

The earliest work performing AD in a functional setting is [Pearlmutter and Siskind 2008]. Their motivations are broader than ours: they want to define a programming language with the ability to perform AD on its own programs. To this end, they endow Scheme with a combinator $\overleftarrow{\mathcal{J}}$ computing the Jacobian of its argument, and whose execution implements reverse mode AD. In order to do this, $\overleftarrow{\mathcal{J}}$ must reflectively access, at runtime, the program in which it is contained, and it must also be possible to apply it to itself. While this offers the possibility of computing higher-order derivatives (in the sense of derivative of the derivative, which we do not consider), it lacks a type-theoretic treatment: the combinator $\overleftarrow{\mathcal{J}}$ is defined in an untyped language. Although [Pearlmutter and Siskind 2008] do mention, for first-order code, a transformation essentially identical to our $\overleftarrow{\mathbf{D}}$ (using so-called *backpropagators*), the observations that backpropagators may be typed linearly and that $\overleftarrow{\mathbf{D}}$ may be directly lifted to higher-order code (as we do in Fig. 5) are original to our work. Finally, let us mention that in this case as well the correctness of $\overleftarrow{\mathcal{J}}$ is not illustrated by a mathematical proof of soundness, but by testing an implementation of it (Stalin∇).

At a more theoretical level, [Elliott 2018] gives a Haskell implementation of backpropagation extracted from a functor $\mathcal{D}$ over cartesian categories, with the benefit of disclosing the compositional nature of the algorithm. However, Elliot's approach is still restricted to first-order programs (*i.e.*, computational graphs): as far as we understand, the functor $\mathcal{D}$ is cartesian but not cartesian closed, so the higher-order primitives (λ-abstraction and application) lack a satisfactory treatment. This is

implicit in Sect. 4.4 of [Elliott 2018], where the author states that he only uses biproduct categories: it is well-known that non-trivial cartesian closed biproduct categories do not exist.[2]

We already mentioned TensorFlow and PyTorch. It is difficult at present to make a fair comparison with such large-scale differentiable programming frameworks since we are still focused on the conceptual level. Nevertheless, in the perspective of a future implementation, our work is interesting because it would offer a way of combining the benefits of hitherto diverging approaches: the ability to generate modular, optimizable code (TensorFlow) with the possibility of using an expressive language for dynamically-generated computational graphs (PyTorch).

Contents of the paper. Sect. 2 gives a (very subjective) introduction to the notions of AD used in this paper. Sect. 3 informally introduces our main contributions, which will then be detailed in the more technical Sect. 5. Sect. 4 formally specifies the programming language we work with, introducing a simply-typed λ-calculus and a rewriting reduction (Fig. 4) necessary to define and to evaluate our backpropagation transformation $\overleftarrow{\mathrm{D}}$. Sect. 6 applies $\overleftarrow{\mathrm{D}}$ to a couple of examples, in particular to a (very simple) recurrent neural network. Sect. 7 concludes with some perspectives.

2 A CRASH COURSE IN AUTOMATIC DIFFERENTIATION

What follows is an introduction to automatic differentiation for those who are not familiar with the topic. It is extremely partial (the field is too vast to be summarized here) and heavily biased not just towards programming languages but towards the very subject of our work. It will hopefully facilitate the reader in understanding the context and achievements of the paper.

2.1 What Is Automatic Differentiation?

Automatic differentiation (or AD) is the science of efficiently computing the derivative of (a restricted class of) programs [Baydin et al. 2017]. Such programs may be represented as directed acyclic hypergraphs, called *computational graphs*, whose nodes are variables of type $\mathbb{R}$ (the set of real numbers) and whose hyperedges are labelled by functions drawn from some finite set of interest (for example, in the context of neural networks, sum, product and some activation function), with the restriction that hyperedges have exactly one target node and that each node is the target of at most one hyperedge. The basic idea is that nodes that are not target of any hyperedge represent input variables, nodes which are not source of any hyperedge represent outputs, and a hyperedge

$$x_1, \dots, x_k \xrightarrow{f} y$$

represents an assignment $y := f(x_1, \dots, x_k)$, so that a computational graph with n inputs and m outputs represents a function of type $\mathbb{R}^n \to \mathbb{R}^m$. An example is depicted in Fig. 1; it represents the function $(x_1, x_2) \mapsto \sin((x_1 - x_2)^2)$.

In terms of programming languages, we may define computational graphs as generated by

$$F,\ G \ ::= \ \mathtt{x} \ \mid \ f(\mathtt{x}_1,\dots,\mathtt{x}_k) \ \mid \ \mathtt{let}\ \mathtt{x} = G\ \mathtt{in}\ F \ \mid \ (F,G)$$

where $\mathtt{x}$ ranges over ground variables and f over a set of real function symbols. The `let` binders are necessary to represent *sharing*, a fundamental feature of hypergraphs: *e.g.*, in Fig. 1, node z_1 is shared. The set of real function symbols consists of one nullary symbol for every real number plus finitely many non-nullary symbols for actual functions. We may write $f(G_1, \dots, G_n)$ as syntactic sugar for $\mathtt{let}\ \mathtt{x}_1 = G_1\ \mathtt{in} \dots \mathtt{let}\ \mathtt{x}_n = G_n\ \mathtt{in}\ f(\mathtt{x}_1, \dots, \mathtt{x}_n)$. Within this introduction (Sect. 2 and 3) we adopt the same notation for a function and the symbol associated with it in the language,

[2] If a category $\mathbf{C}$ has biproducts, then $1 \cong 0$ (the terminal object is also initial). If $\mathbf{C}$ is cartesian closed, then its product $\times$ is a left adjoint and therefore preserves colimits (initial objects in particular), so $A \cong A \times 1 \cong A \times 0 \cong 0$ for every object A of $\mathbf{C}$.

so for example r may refer to both a real number and its associated numeral. Also, we use an `OCaml`-like syntax in the hope that it will help the unacquainted reader parse the examples. From Sect. 4 onward we will adopt a more succinct syntax.

Typing is as expected: types are of the form R^k and every computational graph in context $\mathtt{x}_1 : \mathsf{R}, \ldots, \mathtt{x}_n : \mathsf{R} \vdash G : \mathsf{R}^m$ denotes a function $[\![G]\!] : \mathbb{R}^n \to \mathbb{R}^m$ (this will be defined formally in Sect. 5.1). Restricting for simplicity to the one-output case, we may say that, as far as we are concerned, *the central question of AD is computing the gradient of* $[\![G]\!]$ *at any point* $\mathbf{r} \in \mathbb{R}^n$*, as efficiently as possible*. We remind that the gradient of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ is defined to be the function $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ such that, for all $\mathbf{r} \in \mathbb{R}$,

$$\nabla f(\mathbf{r}) = (\partial_1 f(\mathbf{r}), \ldots, \partial_n f(\mathbf{r})), \tag{1}$$

where by $\partial_i f$ we denote the partial derivative of f with respect to its i-th argument. Of course, the above question makes sense only if $[\![G]\!]$ is differentiable, which is the case if every function symbol represents a differentiable function. In practice this is not always guaranteed (notoriously, modern neural networks use activation functions which are *not* differentiable) but this is not actually an issue, as it will be explained momentarily.

Before delving into the main AD algorithms, let us pause a moment on the question of evaluating a computational graph. In terms of hypergraphs, we are given a computational graph G with input nodes $x_1, \ldots, x_n$ together with an assignment $x_i := r_i$ with $r_i \in \mathbb{R}$ for all $1 \le i \le n$. The value $[\![G]\!](r_1, \ldots, r_n)$ is found by progressively computing the assignments $w := f(s_1, \ldots, s_m)$ for each hyperedge $z_1, \ldots, z_m \xrightarrow{f} w$ such that the values of all z_i are already known. This is known as *forward evaluation* and has cost $|G|$ (the number of nodes of G).

In terms of programming languages, forward evaluation corresponds to a standard call-by-value strategy, with values being defined as tuples of numbers. For instance, for G in Fig. 1

```
                    let x1 = 5 in              let x1 = 5 in              let x1 = 5 in
let x1 = 5 in       let x2 = 2 in              let x2 = 2 in              let x2 = 2 in
let x2 = 2 in  →*   let x1 = 3 in        →*    let x1 = 3 in        →*    let x1 = 3 in
  G                 let z2 = z1 · z1 in        let z2 = 9 in              let z2 = 9 in
                      sin z2                     sin z2                     0.412
```

The operational semantics realizing the above computation will be introduced later. For now, let us mention that the value of a closed computational graph G may be found in $O(|G|)$ reduction steps, where $|G|$ is the size of G as a term, consistently with the cost of forward evaluation.

2.2 Forward Mode AD

The simplest AD algorithm is known as *forward mode* differentiation. Suppose that we are given a computational graph (in the sense of a hypergraph) G with input nodes $x_1, \ldots, x_n$ and one output node y, and suppose that we want to compute its j-th partial derivative in $\mathbf{r} = (r_1, \ldots, r_n) \in \mathbb{R}^n$. The algorithm maintains a memory consisting of a set of assignments of the form $x := (s, t)$, where x is a node of G and $s, t \in \mathbb{R}$ (know as primal and tangent), and proceeds as follows:

- we initialize the memory with $x_i := (r_i, 0)$ for all $0 \le i \le n$, $i \ne j$, and $x_j := (r_j, 1)$.

- At each step, we look for a node $z_1, \dots, z_k \xrightarrow{f} w$ such that $z_i = (s_i, t_i)$ is in memory for all $1 \leq i \leq k$ (there is at least one by acyclicity) and w is unknown, and we add to memory

$$w := \left(f(\mathbf{s}) \,,\, \sum_{i=1}^{k} \partial_i f(\mathbf{s}) \cdot t_i \right) \tag{2}$$

  where we used the abbreviation $\mathbf{s} := s_1, \dots, s_m$.

This procedure terminates in a number of steps equal to $|G|$ and one may show, using the chain rule for derivatives (which we will recall in a moment), that at the end the memory will contain the assignment $y := (\llbracket G \rrbracket(\mathbf{r}), \partial_j \llbracket G \rrbracket(\mathbf{r}))$.

Since the arity k of function symbols is bounded, the cost of computing one partial derivative is $O(|G|)$. Computing the gradient requires computing *all* n partial derivatives, giving of a total cost of $O(n\,|G|)$, which is not very efficient since n may be huge (typically, it is the number of weights of a deep neural network, which may well be in the millions).

For example, if G is the computational graph of Fig. 1 and we start with $x_1 := (5, 1), x_2 := (2, 0)$, we obtain $z_1 := (x_1 - x_2, 1 \cdot 1 - 1 \cdot 0) = (3, 1)$, then $z_2 := (z_1 \cdot z_1 \,,\, z_1 \cdot 1 + z_1 \cdot 1) = (9, 6)$ and finally $y := (\sin(z_2), \cos(z_2) \cdot 6) = (0.412, -5.467)$, which is what we expect since $\partial_1 \llbracket G \rrbracket (x_1, x_2) = \cos((x_1 - x_2)^2) \cdot 2(x_1 - x_2)$.

2.3 Symbolic AD

The AD algorithm presented above is purely numerical, but there is also a *symbolic* approach. The basic idea of (forward mode) symbolic AD is to generate, starting from a computational graph G with n input nodes and 1 output node, a computational graph $\overrightarrow{\mathbf{D}}(G)$ with $2n$ input nodes and 2 output nodes such that forward evaluation of $\overrightarrow{\mathbf{D}}(G)$ corresponds to executing forward mode AD on G, *i.e.*, for all $\mathbf{r} = r_1, \dots, r_n \in \mathbb{R}$, the output of $\overrightarrow{\mathbf{D}}(G)(r_1, 0, \dots, r_j, 1, \dots, r_n, 0)$ is $(\llbracket G \rrbracket(\mathbf{r}), \partial_j \llbracket G \rrbracket(\mathbf{r}))$.

From the programming languages standpoint, symbolic AD is interesting because:

(1) it allows one to perform optimizations on $\overrightarrow{\mathbf{D}}(G)$ which would be inaccessible when simply running the algorithm on G (a typical benefit of frameworks like `TensorFlow`);
(2) it opens the way to compositionality, with the advantages mentioned in the introduction;
(3) being a (compositional) program transformation rather than an algorithm, it offers a viewpoint from which AD may possibly be extended beyond computational graphs.

Recently, [Elliott 2018] pointed out how symbolic forward mode AD may be understood in terms of compositionality, thus intrinsically addressing point (2) above. The very same fundamental remark is implicitly used also by [Wang et al. 2019]. Recall that the derivative of a differentiable function $f : \mathbb{R} \to \mathbb{R}$ is the (continuous) function $f' : \mathbb{R} \to \mathbb{R}$ such that, for all $r \in \mathbb{R}$, the map $a \mapsto f'(r) \cdot a$ is the best linear approximation of f around r. Now, differentiable (resp. continuous) functions are closed under composition, so one may wonder whether the operation $(-)'$ is compositional, *i.e.*, whether $(g \circ f)' = g' \circ f'$. This is notoriously not the case: the so-called *chain rule* states that

$$\forall r \in \mathbb{R}, \qquad (g \circ f)'(r) = g'(f(r)) \cdot f'(r). \tag{3}$$

Nevertheless, there is a slightly more complex construction from which the derivative may be recovered and which has the good taste of being compositional. Namely, define, for f as above,

$$\begin{array}{rccc} \mathbf{D}f : & \mathbb{R} \times \mathbb{R} & \longrightarrow & \mathbb{R} \times \mathbb{R} \\ & (r, a) & \mapsto & (f(r), f'(r) \cdot a)). \end{array}$$

We obviously have $\pi_2 \mathbf{D}f(x, 1) = f'(x)$ for all $x \in \mathbb{R}$ (where π_2 is projection on the second component) and we invite the reader to check, using the chain rule itself, that $\mathbf{D}(g \circ f) = \mathbf{D}g \circ \mathbf{D}f$.

The similarity with forward mode AD is not accidental. Indeed, as observed by [Elliott 2018], forward mode symbolic AD may be understood as a compositional implementation of partial derivatives, along the lines illustrated above. Formally, we may consider two term calculi for computational graphs, let us call them $\mathbf{C}$ and $\mathbf{C}'$, defined just as above but based on two different sets of function symbols $\mathcal{F} \subseteq \mathcal{F}'$, respectively, with $\mathcal{F}'$ containing at least sum and product, and equipped with partial functions

$$\partial_i : \mathcal{F} \longrightarrow \mathcal{F}'$$

for each positive integer i such that, for all $f \in \mathcal{F}$ of arity k and for all $1 \leq i \leq k$, $\partial_i f$ is defined and its arity is equal to k. Then, we define a program transformation $\overrightarrow{\mathbf{D}}$ from $\mathbf{C}$ to $\mathbf{C}'$ which, on types, is given by

$$\overrightarrow{\mathbf{D}}(\mathsf{R}) := \mathsf{R} \times \mathsf{R} \qquad\qquad \overrightarrow{\mathbf{D}}(A \times B) := \overrightarrow{\mathbf{D}}(A) \times \overrightarrow{\mathbf{D}}(B),$$

and, on computational graphs,

$$\overrightarrow{\mathbf{D}}(\mathtt{x}) := \mathtt{x} \qquad\qquad \overrightarrow{\mathbf{D}}(\mathtt{let\ x = } G \mathtt{\ in\ } F) := \mathtt{let\ x = } \overrightarrow{\mathbf{D}}(G) \mathtt{\ in\ } \overrightarrow{\mathbf{D}}(F)$$

$$\overrightarrow{\mathbf{D}}((F,G)) := (\overrightarrow{\mathbf{D}}(F), \overrightarrow{\mathbf{D}}(G)) \qquad\qquad \overrightarrow{\mathbf{D}}(f(\mathbf{x})) := \mathtt{let\ x = (z,\ a)\ in\ } (f(\mathbf{z}), \textstyle\sum_{i=1}^{k} \partial_i f(\mathbf{z}) \cdot \mathtt{a}_i)$$

In the last case, f is of arity k and $\mathtt{let\ x = (z,\ a)\ in}\ \ldots$ stands for $\mathtt{let\ x}_1 \mathtt{\ = (z}_1\mathtt{,a}_1\mathtt{)\ in\ \ldots\ let\ x}_k \mathtt{\ = (z}_k\mathtt{,a}_k\mathtt{)\ in\ \ldots}$. Notice how the case $\overrightarrow{\mathbf{D}}(f(\mathbf{x}))$ is the definition of the operator $\mathbf{D}$ mutatis mutandis, considering that now the arity k is arbitrary. More importantly, we invite the reader to compare it to the assignment (2) in the description of the forward mode AD algorithm: they are essentially identical. The following is therefore not so surprising:

Proposition 1. *Suppose that every $f \in \mathcal{F}$ of arity k corresponds to a differentiable function $f : \mathbb{R}^k \to \mathbb{R}$ and that, for all $1 \leq i \leq k$, $\partial_i f$ is the symbol corresponding to its partial derivative with respect to the i-th input. Then, for every computational graph $\mathbf{x} : \mathsf{R} \vdash G : \mathsf{R}$ with n inputs, for all $1 \leq j \leq n$ and for all $\mathbf{r} = r_1, \ldots, r_n \in \mathbb{R}$, we have the call-by-value evaluation*

$$\mathtt{let\ x}_1 \mathtt{\ = (}r_1\mathtt{,0)\ in\ldots\ let\ x}_j \mathtt{\ = (}r_j\mathtt{,1)\ in\ldots\ let\ x}_n \mathtt{\ = (}r_n\mathtt{,0)\ in\ } \overrightarrow{\mathbf{D}}(G) \to^* (\llbracket G \rrbracket(\mathbf{r}), \partial_j \llbracket G \rrbracket(\mathbf{r}))$$

So we obtained what we wanted: evaluating $\overrightarrow{\mathbf{D}}(G)$ in call-by-value is the same as executing the forward mode AD algorithm on G. Moreover, the definition of $\overrightarrow{\mathbf{D}}$ is fully compositional.[3] Indeed, we merely reformulated the transformation $\overrightarrow{\mathcal{D}_x}$ introduced in [Wang et al. 2019]: the computation of the proposition corresponds to that of $\overrightarrow{\mathcal{D}_{x_j}}(G)$.

Symbolic AD also provides us with an alternative analysis of the complexity of the forward mode AD algorithm. Recall that the length of call-by-value evaluation of computational graphs is linear in the size, so the computation of Proposition 1 takes $O(\left|\overrightarrow{\mathbf{D}}(G)\right|)$ steps. Now, by inspecting the definition of $\overrightarrow{\mathbf{D}}$, it is immediate to see that every syntactic construct of G is mapped to a term of bounded size (remember that the arity k is bounded). Therefore, $\left|\overrightarrow{\mathbf{D}}(G)\right| = O(|G|)$, which is exactly the cost of computing one partial derivative in forward mode. In other words, the cost becomes simply the size of the output of the program transformation.

Notice how Proposition 1 rests on a differentiability hypothesis. As mentioned above, this is not truly fundamental. Indeed, observe that $\overrightarrow{\mathbf{D}}(G)$ is *always* defined, independently of whether the

[3] Technically, one may consider the calculi $\mathbf{C}$ and $\mathbf{C}'$ as cartesian 2-multicategories: types are objects, computational graphs with n inputs and one output are n-ary multimorphisms and evaluation paths are 2-arrows. Then, $\overrightarrow{\mathbf{D}}$ is readily seen to be a morphism of cartesian 2-multicategories. We will not develop such categorical considerations in this paper, we will content ourselves with mentioning them in footnotes.

function symbols it uses are differentiable. This is because, in principle, the maps $\partial_i : \mathcal{F} \to \mathcal{F}'$ are arbitrary and the symbol $\partial_i f$ may have *nothing* to do with the i-th partial derivative of the function that the symbol f represents! Less unreasonably, we may consider "mildly" non-differentiable symbols and associate with them "approximate" derivatives: for example, the rectified linear unit $\mathrm{ReLU}(x)$ defined as `if x<0 then 0 else x` may be mapped by ∂_1 to $\mathrm{Step}(x)$ defined as `if x<0 then 0 else 1`, even though technically the latter is not its derivative because the former is not differentiable in 0. Formally, Step is called a *subderivative* of ReLU. Proposition 1 easily extends to subderivatives and, therefore, AD works just as well on non-differentiable functions, computing *subgradients* instead of gradients, which is perfectly acceptable in practice. This remark applies also to backpropagation and to all the results of our paper, although for simplicity we prefer to stick to the more conventional notion of gradient, and thus work under a differentiability hypothesis.

2.4 Reverse Mode AD, or Backpropagation

A more efficient way of computing gradients in the many inputs/one output case is provided by *reverse mode* automatic differentiation, from which the *backpropagation* (often abbreviated as *backprop*) algorithm derives. As usual, we are given a computational graph (seen as a hypergraph) G with input nodes $x_1, \dots, x_n$ and output node y, as well as $\mathbf{r} = r_1, \dots, r_n \in \mathbb{R}$, which is the point where we wish to compute the gradient. The backprop algorithm maintains a memory consisting of assignments of the form $x := (r, \alpha)$, where x is a node of G and $r, \alpha \in \mathbb{R}$ (the primal and the adjoint), plus a boolean flag with values "marked/unmarked" for each hyperedge of G, and proceeds thus:

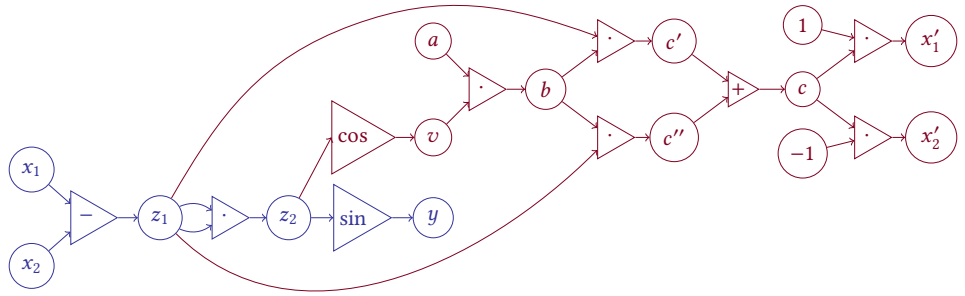
initialization: the memory is initialized with $x_i := (r_i, 0)$ for all $1 \leq i \leq n$, and the *forward phase* starts;

forward phase: at each step, a new assignment $z := (s, 0)$ is produced, with s being computed exactly as during forward evaluation, ignoring the second component of pairs in memory (*i.e.*, s is the value of node z); once every node of G has a corresponding assignment in memory, the assignment $y := (t, 0)$ is updated to $y := (t, 1)$, every hyperedge is flagged as unmarked and the *backward phase* begins (we actually know that $t = [\![G]\!](\mathbf{r})$, but this is unimportant);

backward phase: at each step, we look for an unmarked hyperedge $z_1, \dots, z_k \xrightarrow{f} w$ such that all hyperedges having w among their sources are marked. If no such hyperedge exists, we terminate. Otherwise, assuming that the memory contains $w := (u, \alpha)$ and $z_i := (s_i, \beta_i)$ for all $1 \leq i \leq k$, we update the memory with the assignments $z_i := (s_i\ ,\ \beta_i + \partial_i f(\mathbf{s}) \cdot \alpha)$ (where $\mathbf{s} := s_1, \dots, s_k$) for all $1 \leq i \leq k$ and flag the hyperedge as marked.

One may prove that, when the backprop algorithm terminates, the memory contains assignments of the form $x_i := (r_i, \partial_i [\![G]\!](\mathbf{r}))$ for all $1 \leq i \leq n$, *i.e.*, the value of each partial derivative of $[\![G]\!]$ in $\mathbf{r}$ is computed at the corresponding input node, and thus the gradient may be obtained by just collecting such values. Let us test it on an example. Let G be the computational graph of Fig. 1 and let $r_1 = 5, r_2 = 2$. The forward phase terminates with $x_1 := (5, 0), x_2 := (2, 0), z_1 := (3, 0), z_2 := (9, 0), y := (0.412, 1)$. From here, the backward phase updates $z_2 := (9, \cos(z_2) \cdot 1) = (9, -0.911)$, then $z_1 := (3, z_1 \cdot -0.911 + z_1 \cdot -0.911) = (3, -5.467)$ and finally $x_1 := (5, 1 \cdot -5.467) = (5, -5.467)$ and $x_2 := (2, -1 \cdot -5.467) = (2, 5.467)$, as expected since $\partial_2 [\![G]\!] = -\partial_1 [\![G]\!]$.

Compared to forward mode AD, the backprop algorithm may seem a bit contrived (it is certainly harder to understand why it works) but the gain in terms of complexity is considerable: the forward phase is just a forward evaluation; and, by construction, the backward phase scans each hyperedge exactly once performing each time a constant number of operations. So both phases are linear in $|G|$, giving a total cost of $O(|G|)$, like forward mode. Except that, unlike forward mode, a single evaluation now already gives us the whole gradient, independently of the number of inputs!




```
let z1=x1-x2 in let z2=z1·z1 in let b=(cos z2)·a in let c=z1·b+z1·b in (sin z2,(1·c,-1·c))
```

Fig. 2. The computational graph $\mathbf{bp}_{x_1,x_2,a}(G)$ where G is in Fig. 1, and its corresponding term.

2.5 Symbolic Backpropagation and the Compositionality Issue

The symbolic methodology we saw for forward mode AD may be applied to the reverse mode too: given a computational graph G with n inputs and one output, one may produce a computational graph $\mathbf{bp}(G)$ with $n+1$ inputs and $1+n$ outputs such that, for all $\mathbf{r} = r_1, \ldots, r_n \in \mathbb{R}$, the forward evaluation of $\mathbf{bp}(G)(\mathbf{r}, 1)$ has output $(\llbracket G \rrbracket(\mathbf{r}), \nabla \llbracket G \rrbracket(\mathbf{r}))$. Moreover, $\left|\mathbf{bp}(G)\right| = O(|G|)$.

A formal definition of the **bp** transformation will be given in Sect. 5.1. Let us look at an example, shown in Fig. 2. First of all, observe that $\mathbf{bp}_{x_1,x_2,a}(G)$ contains a copy of G, marked in blue. This corresponds to the forward phase. The nodes marked in red correspond to the backward phase. Indeed, we invite to reader to check that the forward evaluation of $\mathbf{bp}_{x_1,x_2,a}(G)$ with $x_1 := 5$, $x_2 := 2$ and $a := 1$ matches exactly the steps of the backprop algorithm as exemplified in Sect. 2.4, with node b (resp. c, x_1', x_2') corresponding to the second component of the value of node z_2 (resp. z_1, x_1, x_2). (The nodes v, c' and c'' are just intermediate steps in the computation of b and c which are implicit in the numerical description of the algorithm and are hidden in syntactic sugar).

Rather than examining the details of the definition of **bp**, let us observe at once that, from the standpoint of programming languages, it suffers from a serious defect: unlike $\overrightarrow{\mathbf{D}}$, it is *not* compositional. Indeed, in order to define $\mathbf{bp}(\texttt{let x = } G \texttt{ in } F)$, we need to know and exploit the inner structure of $\mathbf{bp}(F)$ and $\mathbf{bp}(G)$, whereas from the definition of $\overrightarrow{\mathbf{D}}$ given above it is clear that no such knowledge is required in that case, *i.e.*, $\overrightarrow{\mathbf{D}}(F)$ and $\overrightarrow{\mathbf{D}}(G)$ are treated as "black boxes". Our way of understanding previous work on the subject [Elliott 2018; Wang et al. 2019] is that it was all about *making symbolic backprop compositional*. This is the topic of the next Section.

3 OUR APPROACH TO COMPOSITIONAL BACKPROPAGATION

As mentioned above, the key to modular and efficient differentiable programming is making symbolic backprop (the **bp** transformation) compositional. We will show that this may be achieved by considering a programming language with a notion of linear negation. The goal of this section is to explain why *negation* and why *linear*.

Let us start with by looking at an extremely simple example, which is just the composition of two unary functions:

$$G := \texttt{let z = } f\ \texttt{x in } g\ \texttt{z} \tag{4}$$

As a hypergraph, G has three nodes x, y, z, of which x is the input and y the output (corresponding to the root of G) and two edges $x \xrightarrow{f} z$ and $z \xrightarrow{g} y$. Note that, since G has only one input, its gradient is equal to its derivative and forward and reverse mode AD have the same complexity. This is why the example is interesting: it shows the difference between the two algorithms by putting them in a context where they must perform essentially the same operations. In the sequel, we set $h := [\![G]\!]$ and we denote by f', g' and h' the derivatives of f, g and h, respectively.

For what concerns forward mode, we invite the reader to check that

$$\overrightarrow{\mathbf{D}}(G) = \texttt{let z = let (v,a)= x in (}f\ \texttt{v, (}f'\ \texttt{v)}\cdot\texttt{a) in let (w, b) = z in (}g\ \texttt{w, (}g'\ \texttt{w)}\cdot\texttt{b)}$$

We may simplify this a bit by observing that (renaming w as z)

$$\overrightarrow{\mathbf{D}}(G)\{\texttt{(x,a)}/\texttt{x}\} \to^* \texttt{let z =}\ f\ \texttt{x in let b = (}f'\ \texttt{x)}\cdot\texttt{a in (}g\ \texttt{z, (}g'\ \texttt{z)}\cdot\texttt{b)}$$

On the other hand, applying the definition of Sect. 5.1, we get

$$\mathbf{bp}_{\texttt{x},\texttt{a}}(G) = \texttt{let z =}\ f\ \texttt{x in let b = (}g'\ \texttt{z)}\cdot\texttt{a in (}g\ \texttt{z,(}f'\ \texttt{x)}\cdot\texttt{b)}$$

Note that, if we substitute $r \in \mathbb{R}$ for x and 1 for a, in both cases we obtain $(h(r), h'(r))$ in the same number of steps, as expected. However, the order in which the derivatives are computed relative to the order of the composition $g \circ f$ is different: it is covariant in forward mode, contravariant in reverse mode. This corresponds precisely to the behavior of the two algorithms:

- in forward mode, we start with $x := (r, 1)$, from which we infer $z := (f(r), f'(r))$, from which we infer $y := (g(f(r)), g'(f(r)) \cdot f'(r))$;
- in reverse mode, the forward phase leaves us with $x := (r, 0)$, $z := (f(r), 0)$, $y := (g(f(r)), 1)$, at which point the backward phase proceeds back from the output towards the input, inferring first $z := (f(r), g'(f(r)))$ and then $x := (r, f'(r) \cdot g'(f(r)))$.

A lesson we learn from this example, in the perspective of compositionality, is that both algorithms may be seen as mapping the subprograms f and g, which have one input and one output, to two subprograms $\overrightarrow{f}$ and $\overrightarrow{g}$ (or $\overleftarrow{f}$ and $\overleftarrow{g}$) having two inputs and two outputs, but then assembling them rather differently:

The picture on the right suggested to [Wang et al. 2019] the idea of solving the compositionality issue via *continuations*: drawing inspiration from "There and Back Again" [Danvy and Goldberg 2005], the blocks $\overleftarrow{f}$ and $\overleftarrow{g}$ are seen as function calls in the CPS transform of G, so that the forward phase takes place along the call path, while the backward phase takes place along the return path. However, in order for their approach to work, [Wang et al. 2019] must use references, *i.e.*, the memory maintained by the backprop algorithm is explicitly present and is manipulated as described in Sect. 2.4. Moreover, since the memory is updated *after* the return from each function call, they must actually resort to *delimited continuations*. On the whole, although they do succeed in presenting reverse mode AD as a compositional program transformation, the work of [Wang et al. 2019] is closer to an (elegant!) implementation of the backprop algorithm in a functional language with references than to an abstract, purely compositional description of its dynamics.

Let us focus, instead, on the idea of contravariance. The archetypal contravariant operation is *negation*. For a (real) vector space A, negation corresponds to the *dual space* $A \multimap \mathbb{R}$, which may be generalized to $A^{\perp_E} := A \multimap E$ for an arbitrary space E, although in fact we will always take $E = \mathbb{R}^d$ for some $d \in \mathbb{N}$. For brevity, let us keep E implicit and simply write $A^\perp$. There is a canonical way, resembling CPS, of transforming a differentiable function $f : \mathbb{R} \to \mathbb{R}$ with derivative f' into a

function $\mathbf{D}_{\mathrm{r}} f : \mathbb{R} \times \mathbb{R}^\perp \to \mathbb{R} \times \mathbb{R}^\perp$ from which the derivative of f may be extracted. Namely, let, for all $x \in \mathbb{R}$ and $x^* \in \mathbb{R}^\perp$,

$$\mathbf{D}_{\mathrm{r}} f(x, x^*) := (f(x), \lambda a.x^*(f'(x) \cdot a)), \tag{5}$$

where we are using λ-notation with the standard meaning. We let the reader verify that, if we suppose $E = \mathbb{R}$, then for all $x \in \mathbb{R}$, $(\pi_2 \mathbf{D}_{\mathrm{r}} f(x, I))1 = f'(x)$, where π_2 is projection of the second component and $I : \mathbb{R} \to \mathbb{R}$ is the identity function (which is obviously linear). More importantly, $\mathbf{D}_{\mathrm{r}}$ is compositional: for all $x \in \mathbb{R}$ and $x^* \in \mathbb{R} \multimap \mathbb{R}$, we have[4]

$$\begin{aligned}\mathbf{D}_{\mathrm{r}} g(\mathbf{D}_{\mathrm{r}} f(x, x^*)) &= \mathbf{D}_{\mathrm{r}} g(f(x), \lambda a.x^*(f'(x) \cdot a)) = (g(f(x)), \lambda b.(\lambda a.x^*(f'(x) \cdot a))(g'(f(x)) \cdot b)) \\ &= (g(f(x)), \lambda b.x^*(f'(x) \cdot (g'(f(x)) \cdot b)) = (g(f(x)), \lambda b.x^*((g'(f(x)) \cdot f'(x)) \cdot b)) \\ &= ((g \circ f)(x), \lambda b.x^*((g \circ f)'(x) \cdot b)) = \mathbf{D}_{\mathrm{r}}(g \circ f)(x, x^*).\end{aligned}$$

This observation may be generalized to maps $f : \mathbb{R}^n \to \mathbb{R}$: for all $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{x}^* = x_1^*, \ldots, x_n^* \in \mathbb{R}^\perp$,

$$\overleftarrow{\mathbf{D}}(f)(\mathbf{x}; \mathbf{x}^*) := \left(f(\mathbf{x}), \lambda a. \sum_{i=1}^{n} x_i^* \, (\partial_i f(\mathbf{x}) \cdot a) \right).$$

In the AD literature, the x_i^* are called *backpropagators* [Pearlmutter and Siskind 2008]. Obviously $\overleftarrow{\mathbf{D}}(f) : (\mathbb{R} \times \mathbb{R}^\perp)^n \to \mathbb{R} \times \mathbb{R}^\perp$ and we invite the reader to check that, if we take $E = \mathbb{R}^n$, we have

$$(\pi_2 \overleftarrow{\mathbf{D}}(f)(\mathbf{x}; \iota_1, \ldots, \iota_n))1 = \nabla f(\mathbf{x}),$$

where, for all $1 \le i \le n$, $\iota_i : \mathbb{R} \to \mathbb{R}^n$ is the injection into the i-th component, *i.e.*, $\iota_i(x) = (0, \ldots, x, \ldots, 0)$ with zeros everywhere except at position i, which is a linear function. Moreover, $\overleftarrow{\mathbf{D}}$ is compositional.[5] This leads to the definition of a compositional program transformation $\overleftarrow{\mathbf{D}}$ (Fig. 5) which verifies $\overleftarrow{\mathbf{D}}(\mathsf{R}) = \mathsf{R} \times \mathsf{R}^\perp$ and which, applied to our example (4), gives

$$\overleftarrow{\mathbf{D}}(G) = \begin{array}{l}\texttt{let z =} \\ \quad \texttt{let (v,v*) = x in} \\ \quad \texttt{(}f\texttt{ v, fun b -> v* ((}f'\texttt{ v) · b)) in} \\ \texttt{let (w, w*) = z in} \\ \texttt{(}g\texttt{ w, fun a -> w* (}g'\texttt{(w) · a))}\end{array}$$

where $\mathtt{w}^*, \mathtt{v}^* : \mathsf{R}^\perp$ so that both $\mathtt{fun\ b\ ->\ v^*\ ((}f'\mathtt{\ v)\ \cdot\ b))}$ and $\mathtt{fun\ a\ ->\ w^*\ ((}g'\mathtt{\ w)\ \cdot\ a)}$ have also type $\mathsf{R}^\perp$. Notice the resemblance of $\overleftarrow{\mathbf{D}}(G)$ and $\overrightarrow{\mathbf{D}}(G)$: this is not an accident, both are defined in a purely compositional way (in particular, $\overleftarrow{\mathbf{D}}(\mathtt{let\ x\ =}\ H\ \mathtt{in}\ F) = \mathtt{let\ x\ =}\ \overleftarrow{\mathbf{D}}(H)\ \mathtt{in}\ \overleftarrow{\mathbf{D}}(F)$), abiding by the "black box" principle) and the only non trivial case is when they are applied to function symbols. Moreover, we have

$$\overleftarrow{\mathbf{D}}(G)\{\mathtt{(x,x^*)/x}\} \to^* \mathtt{let\ z\ =}\ f\ \mathtt{x\ in\ (}g\ \mathtt{z,\ fun\ a\ ->\ let\ b\ =\ (}g'\ \mathtt{z)\cdot a\ in\ x^*((}f'\ \mathtt{x)\cdot b))}$$

[4] The equation in the second line uses both commutativity and associativity of product, but only the latter is really necessary: by replacing $f'(x) \cdot a$ with $a \cdot f'(x)$ in Eq. 5 one can check that commutativity is not needed. The former notation reflects that this is actually a linear application: in general, if $f : A \to B$, then $f'(x) : A \multimap B$ and $a : A$. When $A = B = k$ with k a ring (commutative or not), $k \multimap k \cong k$ and linear application becomes the product of k, so the notation $a \cdot f'(x)$ makes sense and backprop has in fact been applied to non-commutative rings [Isokawa et al. 2003; Pearson and Bisset 1992]. In the general case, it makes no sense to swap function and argument and it is unclear how backprop would extend.

[5] Technically, functions of type $\mathbb{R}^n \to \mathbb{R}$ for varying n form what is known as a *cartesian operad*, or *clone*, and $\overleftarrow{\mathbf{D}}$ is a morphism of such structures. A bit more explicitly, one may compose $f : \mathbb{R}^n \to \mathbb{R}$ with $g : \mathbb{R}^{m+1} \to \mathbb{R}$ by "plugging" f into the i-th coordinate of g, forming $g \circ^i f : \mathbb{R}^{n+m} \to \mathbb{R}$, for any $1 \le i \le m + 1$; the operation $\overleftarrow{\mathbf{D}}$ preserves such compositions.

which, albeit of different type, is essentially identical to $\mathbf{bp}_{x,a}(G)$. More precisely, if we write $\texttt{let b = } (g'\ \texttt{z})\cdot\texttt{a in } (f'\ \texttt{x})\cdot\texttt{b}$ as F, then we have

$$\mathbf{bp}_{x,a}(G) = \texttt{let z=}f\ \texttt{x in } (g\ \texttt{z}, F) \text{ and } \overleftarrow{\mathbf{D}}(G)\,\{(\texttt{x}, \texttt{x}^*)/\texttt{x}\} \to^* \texttt{let z=}f\ \texttt{x in } (g\ \texttt{z},\ \texttt{fun a -> x}^*\ F)$$

Let us now explain why negation must be *linear*. The above example is too simple for illustrating this point, so from now on let G be the computational graph of Fig. 1. Applying the definition of Fig. 5 and simplifying, we obtain that $\overleftarrow{\mathbf{D}}(\texttt{G})$ is equal to:

```
let z2 =
   let z1 =
      let (v2, v2*) = x2 in
      let (v1, v1*) = x1 in
      (v1-v2, fun c -> v1*(1·c)+v2*(−1·c)) in
   let (w1,w1*) = z1 in
   (w1·w1, fun b -> w1*(w1·b)+w1*(w1·b)) in
let (w2, w2*) = z2 in
(sin w2, fun a -> w2* ((cos w2)·a))
```

$\longrightarrow^*$

```
let (v2, v2*) = x2 in
let (v1, v1*) = x1 in
let (z1,z1*) =
   (v1-v2, fun c -> v1*(1·c)+v2*(−1·c)) in
let (z2, z2*) =
   (z1·z1, fun b -> z1*(z1·b)+z1*(z1·b)) in
(sin z2, fun a -> z2* ((cos z2)·a))
```

There is a potential issue here, due to the presence of two occurrences of $\texttt{z}_1^*$ (highlighted in brown) which are matched against the function corresponding to the derivative of $\texttt{x}_1\texttt{-x}_2$ (also highlighted in brown, let us denote it by F). Notice that such a derivative is present *only once* in $\mathbf{bp}_{x_1,x_2,a}(G)$: it corresponds to the rightmost nodes of Fig. 2 (more precisely, the abstracted variable c corresponds to node c itself, whereas $\texttt{v}_1^*$ and $\texttt{v}_2^*$ correspond to nodes x_1' and x_2', respectively). Therefore, duplicating F would be a mistake in terms of efficiency.

The key observation here is that $\texttt{z}_1^*$ is of type $\mathsf{R}^\perp$, *i.e.*, it is a *linear* function (and indeed, F is linear: by distributivity of product over sum, c morally appears only once in its body). This means that, for all $t, u : \mathsf{R}$, we have $z_1^* t + z_1^* u = z_1^*(t+u)$. In the λ-calculus we consider (Sect. 4), this becomes an evaluation step oriented from left to right, called *linear factoring* (18), allowing us to evaluate $\overleftarrow{\mathbf{D}}(G)$ with the same efficiency as $\mathbf{bp}_{x_1,x_2,a}(G)$. The linear factoring $ft + fu \to f(t+u)$ would be semantically unsound in general if $f : \neg\mathsf{R} = \mathsf{R} \to \mathsf{R}$, which is why we must explicitly track the linearity of negations.

So we have a compositional transformation $\overleftarrow{\mathbf{D}}$ which takes a computational graph G with n inputs $x_1, \ldots, x_n$ and returns a program $x_1 : \mathsf{R} \times \mathsf{R}^\perp, \ldots, x_n : \mathsf{R} \times \mathsf{R}^\perp \vdash \overleftarrow{\mathbf{D}}(G) : \mathsf{R} \times \mathsf{R}^\perp$ in the simply-typed λ-calculus augmented with linear negation, such that $\overleftarrow{\mathbf{D}}(G)$ evaluates to (essentially) $\mathbf{bp}_{x_1,\ldots,x_n,a}(G)$. Actually, thanks to another nice observation of [Wang et al. 2019], we can do much more: we can extend $\overleftarrow{\mathbf{D}}$ for free to the *whole* simply-typed λ-calculus, just letting $\overleftarrow{\mathbf{D}}(\texttt{fun x ->}t) := \texttt{fun x ->}\overleftarrow{\mathbf{D}}(t)$ and $\overleftarrow{\mathbf{D}}(tu) := \overleftarrow{\mathbf{D}}(t)\overleftarrow{\mathbf{D}}(u)$, and, whenever $\texttt{x}_1 : \mathsf{R}, \ldots, \texttt{x}_n : \mathsf{R} \vdash t : \mathsf{R}$, we have that $\overleftarrow{\mathbf{D}}(t)$ still computes $\nabla\,[\![t]\!]$ with the same efficiency as the evaluation of t! Indeed, the definition of $\overleftarrow{\mathbf{D}}$ immediately gives us that if $t \to^* u$ in p steps, then $\overleftarrow{\mathbf{D}}(t) \to^* \overleftarrow{\mathbf{D}}(u)$ in $O(p)$ steps (point 2 of Lemma 13).[6] But since t has ground type and ground free variables, eliminating all higher-order redexes gives $t \to^* G$ for some computational graph G, hence $\overleftarrow{\mathbf{D}}(t)$ evaluates to (essentially) $\mathbf{bp}(G)$. So $\overleftarrow{\mathbf{D}}(t)$ computes the gradient of $[\![t]\!] = [\![G]\!]$ (remember that the semantics is invariant under evaluation) with a cost equal to $O(|G|)$ plus the cost of the evaluation $t \to^* G$, which is the best we can expect in general.

[6] Morally, the simply-typed λ-calculus augmented with a set $\mathcal{F}$ of function symbols is the free cartesian semi-closed 2-multicategory on $\mathcal{F}$ (semi-closed in the sense of [Hyland 2017]). Therefore, once $\overleftarrow{\mathbf{D}}$ is defined on $\mathcal{F}$, it automatically extends to a morphism of such structures. In particular, it functorially maps evaluations (which are 2-arrows) to evaluations.

To conclude, we should add that in the technical development we actually use Accattoli's *linear substitution calculus* [Accattoli 2012], which is a bit more sophisticated than the standard simply-typed λ-calculus. This is due to the presence of linear negation, but it is also motivated by efficiency, which is the whole point of backpropagation. To be taken seriously, a proposal of using functional programming as the foundation of (generalized) AD ought to come with a thorough complexity analysis ensuring that we are not losing efficiency in moving from computational graphs to λ-terms. Thanks to its tight connection with abstract machines and reasonable cost models [Accattoli et al. 2014], ultimately owed to its relationship with Girard's proof nets [Accattoli 2018] (a graphical representation of linear logic proofs which may be seen as a higher order version of computational graphs), the linear substitution calculus is an ideal compromise between the abstractness of the standard λ-calculus and the concreteness of implementations, and provides a solid basis for such an analysis.

4 THE LINEAR SUBSTITUTION CALCULUS

Terms and Types. Since the linear factoring rule (18) is type-sensitive (as mentioned above, it is unsound in general), it is convenient to adopt a Church-style presentation of our calculus, *i.e.*, with explicit type annotations on variables. The set of *types* is generated by the following grammar:

$$A, B, C ::= \mathsf{R} \mid A \times B \mid A \to B \mid \mathsf{R}^{\perp_d} \qquad \text{(simple types)}$$

where R is the ground type of real numbers. The negation $\mathsf{R}^{\perp_d}$ corresponds to the linear implication (in the sense of linear logic [Girard 1987]) $\mathsf{R} \multimap \mathsf{R}^d$. However, in order to keep the calculus as simple as possible, we avoid introducing a fully-fledged bilinear application (as for example in the bang-calculus [Ehrhard and Guerrieri 2016]) and opt instead for just a negation operator and dedicated typing rules. We may omit the subscript d in $\mathsf{R}^{\perp_d}$ if clear from the context or unimportant.

An *annotated variable* is either $x^{!A}$ (called *exponential variable*) with A any type, or x^{R} (called *linear variable*): the writing $x^{(!)A}$ stands for one of the two annotations (in the linear case $A = \mathsf{R}$). The grammar of *values* and *terms* is given by mutual induction as follows, with $x^{(!)A}$ varying over the set of annotated variables, r over the set of real numbers $\mathbb{R}$ and f over a finite set $\mathcal{F}$ of function symbols over real numbers containing at least multiplication (noted in infix form $t \cdot u$):

$$v ::= x^{(!)A} \mid \underline{r} \mid \lambda x^{(!)A}.t \mid \langle v_1, v_2 \rangle \qquad \text{(values)} \qquad (6)$$

$$t, u ::= v \mid tu \mid \langle t, u \rangle \mid t[\langle x^{!A}, y^{!B} \rangle := u] \mid t[x^{(!)A} := u] \mid t + u \mid f(t_1, \dots, t_k) \qquad \text{(terms)} \qquad (7)$$

Since binders contain type annotations, bound variables will never be annotated in the sequel. In fact, we will entirely omit type annotations if irrelevant or clear from the context. Terms of the form $\underline{r}$ are called *numerals*. The term $t[x := u]$ (and its binary version $t[\langle x, y \rangle := u]$) may be understood as the more familiar `let x = u in t` used in the previous informal sections.

We denote by $\Lambda_\perp(\mathcal{F})$ the set of terms generated by the above grammar. We consider also the subset $\Lambda(\mathcal{F})$ of terms obtained by discarding linear negation and linear variables.

We denote by $|t|$ the *size* of a term t, *i.e.*, the number of symbols appearing in t. We denote by $\mathrm{fv}(t)$ the set of *free variables* of t, abstractions and explicit substitutions being the binding operators. As usual, α-equivalent terms are treated as equal. A term t is *closed* if $\mathrm{fv}(t) = \emptyset$. In the following, we will use boldface metavariables to denote sequences: $\mathbf{x}$ will stand for a sequence of variables $x_1, \dots, x_n$, $\mathbf{t}$ for a sequence of terms $t_1, \dots, t_n$, etc. The length of the sequences will be left implicit, because either clear from the context or irrelevant.

We use n-ary products $\langle t_1, \dots, t_{n-1}, t_n \rangle$ as syntactic sugar for $\langle t_1, \dots \langle t_{n-1}, t_n \rangle \dots \rangle$, and we define *Euclidean types* by $\mathsf{R}^d := \mathsf{R} \times (\dots \mathsf{R} \times \mathsf{R} \dots)$. It will be useful to denote a bunch of sums without specifying the way these sums are associated. The notation $\sum_{i \in I} t_i$, will denote such a bunch for I

$$\frac{}{\Gamma \vdash_z z : \mathrm{R}} \qquad \frac{}{\Gamma, x^{!A} \vdash x : A} \qquad \frac{\Gamma \vdash_{(z)} t : A \quad \Gamma \vdash_{(z)} u : B}{\Gamma \vdash_{(z)} \langle t, u \rangle : A \times B} \qquad \frac{\Gamma \vdash u : A \times B \quad \Gamma, x^{!A}, y^{!B} \vdash_{(z)} t : C}{\Gamma \vdash_{(z)} t[\langle x^{!A}, y^{!B} \rangle := u] : C}$$

$$\frac{\Gamma, x^{!A} \vdash t : B}{\Gamma \vdash \lambda x^{!A}.t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B} \qquad \frac{\Gamma \vdash_z t : \mathrm{R}^d}{\Gamma \vdash \lambda z^{\mathrm{R}}.t : \mathrm{R}^{\perp d}} \qquad \frac{\Gamma \vdash t : \mathrm{R}^{\perp d} \quad \Gamma \vdash_{(z)} u : \mathrm{R}}{\Gamma \vdash_{(z)} tu : \mathrm{R}^d}$$

$$\frac{\Gamma \vdash u : A \quad \Gamma, x^{!A} \vdash_{(z)} t : C}{\Gamma \vdash_{(z)} t[x^{!A} := u] : C} \qquad \frac{\Gamma \vdash_{(z')} u : \mathrm{R} \quad \Gamma \vdash_z t : \mathrm{R}^d}{\Gamma \vdash_{(z')} t[z^{\mathrm{R}} := u] : \mathrm{R}^d} \qquad \frac{\Gamma \vdash t_1 : \mathrm{R} \quad \dots \quad \Gamma \vdash t_k : \mathrm{R}}{\Gamma \vdash f(t_1, \dots, t_k) : \mathrm{R}} \qquad \frac{r \in \mathbb{R}}{\Gamma \vdash \underline{r} : \mathrm{R}}$$

$$\frac{\Gamma \vdash_{(z)} t : \mathrm{R} \quad \Gamma \vdash u : \mathrm{R}}{\Gamma \vdash_{(z)} t \cdot u : \mathrm{R}} \qquad \frac{\Gamma \vdash t : \mathrm{R} \quad \Gamma \vdash_{(z)} u : \mathrm{R}}{\Gamma \vdash_{(z)} t \cdot u : \mathrm{R}} \qquad \frac{}{\Gamma \vdash_z \underline{0} : \mathrm{R}^d} \qquad \frac{\Gamma \vdash_{(z)} t : \mathrm{R}^d \quad \Gamma \vdash_{(z)} u : \mathrm{R}^d}{\Gamma \vdash_{(z)} t + u : \mathrm{R}^d}$$

Fig. 3. The typing rules. In the pairing and sum rules, either all three sequents have z, or none does.

a finite set. In case I is a singleton, the sum denotes its unique element. An empty sum of type R^d stands for $\langle \underline{0}, \dots, \underline{0} \rangle$, which we denote by $\underline{\mathbf{0}}$. Of course this notation would denote a unique term modulo associativity and commutativity of + and neutrality of $\underline{0}$, but we do not need to introduce these equations in the calculus.

The typing rules are in Fig. 3. The meta-variables Γ, Δ vary over the set of typing contexts, which are finite sequences of exponential type annotated variables without repetitions. There are two kind of sequents: $\Gamma \vdash t : A$ and $\Gamma \vdash_z t : \mathrm{R}^d$. In this latter $d \in \mathbb{N}$ and z is linear type annotated variable which occurs free *linearly* in t. The typing rules define what "occurring linearly" means, following the standard rules of linear logic.[7] The writing $\Gamma \vdash_{(z)} t : A$ stands for either $\Gamma \vdash t : A$ or $\Gamma \vdash_z t : A$, and in the latter case $A = \mathrm{R}^d$ for some d.

Contexts. We consider one-hole contexts, or simply *contexts*, which are defined by the above grammar (6) restricted to the terms having exactly one occurrence of a specific variable $\{\cdot\}$, called the *hole*. Meta-variables C, D will range over the set of one-hole contexts. Given a context C and a term t we denote by $\mathrm{C}\{t\}$ the substitution of the hole in C by t allowing the possible capture of free variables of t. A particular class of contexts are the *substitution contexts* which have the form of a hole followed by a stack (possibly empty) of explicit substitutions: $\{\cdot\}[p_1 := t_1] \dots [p_n := t_n]$ with each p_i a variable or a pair of variables. Meta-variables α, β will range over substitution contexts. If α is a substitution context, we will use the notation $t\alpha$ instead of $\alpha\{t\}$.

Rewriting Rules. The *reduction relation* is given in Fig. 4 divided in three sub-groups:

$\beta := \{(8), (9), (10), (11), (12), (13), (14), (15)\}$	evaluation rules,
$\eta := \{(16), (17)\}$	extensional rules,
$\ell := \{(18)\}$	linear factoring.

In case one wants to consider numeric computations (other than sum and products), then of course one must also include suitable reduction rules associated with the function symbols:

$$f(\underline{r}_1 \alpha_1, \dots, \underline{r}_n \alpha_n) \to \underline{[\![f]\!](r_1, \dots, r_n)} \alpha_1 \dots \alpha_n \tag{24}$$

[7] In this paper we focus on exactly what is required to express the backpropagation algorithm in the λ-calculus, avoiding a full linear logic typing assignment and just tracking the linearity of a single variable of type R in judgments typing a term with a Euclidean type R^d.

$$(\lambda x.t)\alpha u \to t[x := u]\alpha \tag{8}$$
$$s[\langle x, y\rangle := \langle t, u\rangle\, \alpha] \to s[x := t][y := u]\alpha \tag{9}$$
$$\mathrm{C}\{x\}[x^{!A} := v\alpha] \to \mathrm{C}\{v\}[x^{!A} := v]\alpha \tag{10}$$
$$t[x^{!A} := v\alpha] \to t\alpha \qquad \text{if } x \notin \mathrm{fv}(t) \tag{11}$$
$$t[x^{\mathrm{R}} := v\alpha] \to t\{v/x\}\,\alpha \tag{12}$$
$$\underline{r}\alpha + \underline{q}\beta \to \underline{r + q}\alpha\beta \tag{13}$$
$$\langle t_1, t_2\rangle\,\alpha + \langle u_1, u_2\rangle\,\beta \to \langle t_1 + u_1, t_2 + u_2\rangle\,\alpha\beta \tag{14}$$
$$\underline{r}\alpha \cdot \underline{q}\beta \to \underline{rq}\alpha\beta \tag{15}$$

(a) β-rules. In (10), (11) and (12), v is a value. In (10), C is an arbitrary context not binding x. We write $(10)^{\mathrm{n}}$ to refer to the instance of (10) in which v is a numeral.

$$t \to \lambda y.ty \tag{16}$$
$$t[x := \langle u, u'\rangle] \to t[x := \langle y, y'\rangle][\langle y, y'\rangle := \langle u, u'\rangle] \tag{17}$$

(b) η-rules. In (16) t has an arrow type or $\mathrm{R}^{\perp}$. The new variables on the right-hand side of both rules are fresh.

$$(x^{\mathrm{R}^{\perp}}\alpha t)\beta + (x^{\mathrm{R}^{\perp}}\alpha' t')\beta' \to x^{\mathrm{R}^{\perp}}(t + t')\alpha\beta\alpha'\beta' \tag{18}$$

(c) linear factoring (ℓ-rule for short), where we suppose that none of $\alpha, \beta, \alpha', \beta'$ binds x.

$$t[x := u][y := w] \equiv t[y := w][x := u] \qquad \text{if } y \notin \mathrm{fv}(u) \text{ and } x \notin \mathrm{fv}(w) \tag{19}$$
$$t[x := u][y := w] \equiv t[x := u[y := w]] \qquad \text{if } y \notin \mathrm{fv}(t) \tag{20}$$
$$t[x^{!A} := u] \equiv t_{\{y/x\}}[x^{!A} := u][y^{!A} := u] \tag{21}$$
$$(s\Box t)[x := u] \equiv s[x := u]\Box t \qquad \text{if } x \notin \mathrm{fv}(t) \tag{22}$$
$$(s\Box t)[x := u] \equiv s\Box(t[x := u]) \qquad \text{if } x \notin \mathrm{fv}(s) \tag{23}$$

(d) Structural equivalence. In (21), $t_{\{y/x\}}$ denotes t in which some (possibly none) occurrences of x are renamed to a fresh variable y. In (22), (23) the writing $s\Box t$ stands for either st or $s + t$ or $s \cdot t$ or $\langle s, t\rangle$.

Fig. 4. The reduction and the structural equivalence relations, where we suppose the usual convention that no free variable in one term can be captured in the other term of a relation.

The rule (8) transforms a λ-calculus application into an explicit substitution. The difference between the two is that the latter commutes over terms by the structural equivalence defined in Fig. 4d, while the former does not. Rule (9) deconstructs a pair, while rule (10) implements a "micro-step" substitution, closer to abstract machines [Accattoli et al. 2014]. The special case in which v is a numeral is referred to as $(10)^{\mathrm{n}}$. Rule (11) implements garbage collection, and rule (12) linear substitution. The rules (16) and (17) are standard instances of η-expansion rules. They are useful in the proof of Theorem 15. Rule (18) has already been discussed.

Notice that $\Lambda(\mathcal{F})$ is a standard linear explicit substitution calculus encompassing both call-by-need and call-by-value [Accattoli et al. 2014]. For instance, the usual by-value β-rule $(\lambda x.t)v \to t\{v/x\}$ is derivable. In this respect, the reader may think of $\Lambda(\mathcal{F})$ as nothing but the plain simply-typed λ-calculus, and consider explicit substitutions as needed merely to represent computational

graphs (which are obtained by restricting to the ground type R only). The situation is different in $\Lambda_\perp(\mathcal{F})$, in which linearity plays a key role for expressing backpropagation.

Given any $X \subseteq \beta \cup \eta \cup \ell$, we denote by $\xrightarrow{X}$ the context closure of the union of the reduction relations in X, for any context C:

$$\mathrm{C}\{t\} \xrightarrow{X} \mathrm{C}\{u\}, \text{ whenever } t \xrightarrow{X} u.$$

This means that we do not consider a specific evaluation strategy (call-by-value, call-by-name etc…), in order to be as general as possible and to allow a future analysis concerning a more efficient operational semantics.

A term t is a *X-normal form* if there is no term u with $t \xrightarrow{X} u$. If the set X is a singleton $\{\iota\}$, we simply write $\xrightarrow{\iota}$. If $X = \beta \cup \eta \cup \ell$, we write just $\to$. If $k \in \mathbb{N}$, $\xrightarrow{X}{}^{k}$ denotes a sequence of length k of $\xrightarrow{X}$, whereas $\xrightarrow{X}{}^{*}$ denotes a sequence arbitrary length (including null), *i.e.*, $\xrightarrow{X}{}^{*}$ is the reflexive-transitive closure of $\xrightarrow{X}$. Juxtaposition of labels means their union, so that, for example, $\xrightarrow{\beta\eta}$ denotes the context closure of all reduction relations except (18).

Structural equivalence $\equiv$ is the smallest equivalence relation containing the context closure of the rules (19)–(23) in Fig. 4d. Morally, structural equivalence relates terms which would correspond to the same state of an abstract machine implementing the calculus, in which explicit substitutions represent pointers to memory. We refer to [Accattoli et al. 2014; Accattoli and Barras 2017] for more details. The crucial property of $\equiv$ is that it is a (strong) bisimulation with respect to $\to$, which means in particular that it may always be postponed to the end of an evaluation (Proposition 4). The following properties are standard and we give them without proof.

Proposition 2 (Subject reduction). *If $t \to u$ or $t \equiv u$ and $\Gamma \vdash_{(z)} t : A$, then $\Gamma \vdash_{(z)} u : A$.*

Lemma 3 ($\equiv$ is a strong bisimulation). *Let ι be any reduction rule and let $t' \equiv t \xrightarrow{\iota} u$, then there exists $t' \xrightarrow{\iota} u'$ such that $u' \equiv u$.*

Proposition 4 (postponement of $\equiv$). *Let X be any subset of the reduction rules in Fig. 4 (including the variant $(10)^{\mathrm{n}}$) and let t $(\xrightarrow{X} \cup \equiv)^{*}$ u with k X-steps, then there exists u' such that $t \xrightarrow{X}{}^{k} u' \equiv u$.*

Proposition 5 (values). *Given a closed term t of type A, if t is a β-normal form, then it is a value.*

Proposition 6 (Weak normalization). *For every term t, and every set $X \subseteq \beta\ell$, there exists a X-normal form u such that $t \xrightarrow{X}{}^{*} u$.*

The $\beta\ell$-rewriting enjoys also strong normalization, even modulo $\equiv$, but the proof is more involved and uninteresting for our purposes, so we omit it. The strong normalization is however immediate if we restrict the contraction rule (10) to numerals, a property which will be useful in the sequel.

Lemma 7. *For any $t \xrightarrow{(10)^{\mathrm{n}}(11)(14)(13)(15)}{}^{*} u$, the number of steps in the sequence is $O(|t|)$.*

Denotational Semantics. The cartesian closed category of sets and functions gives a denotational model for this calculus. Types are interpreted as sets, as follows:

$$[\![\mathrm{R}]\!] := \mathbb{R} \qquad [\![A \to B]\!] := \text{set of functions from } [\![A]\!] \text{ to } [\![B]\!]$$
$$[\![A \times B]\!] := [\![A]\!] \times [\![B]\!] \qquad [\![\mathrm{R}^{\perp_d}]\!] := \text{set of linear maps from } \mathbb{R} \text{ to } \mathbb{R}^d$$

Notice that the restriction to linear functions in $[\![\mathrm{R}^{\perp_d}]\!]$ is such that rule (18) is sound. The interpretation of a judgment $\Gamma \vdash t : A$ is a function $[\![t]\!]^{\Gamma}$ from the cartesian product $[\![\Gamma]\!]$ of the denotations

of the types in Γ to $[\![A]\!]$. The interpretation of a judgment $\Gamma \vdash_z t : \mathrm{R}^d$, is given as a function $[\![t]\!]^{\Gamma;z}$ associating with every $\mathbf{g} \in [\![\Gamma]\!]$ a linear map from $\mathbb{R}$ to $\mathbb{R}^d$. The definition is by induction on t and completely standard (explicit substitution is functional composition). We omit the superscript Γ (or $\Gamma; z$) if irrelevant. This interpretation supposes to have associated each function symbol in $\mathcal{F}$ with a suitable map over real numbers. By a standard reasoning, one can prove that:

Proposition 8 (Semantic soundness). *Let $\Gamma \vdash_{(z)} t : A$, then $t \to u$ or $t \equiv u$, gives $[\![t]\!] = [\![u]\!]$.*

The semantic soundness gives as a by-product a light form of confluence on Euclidean types[8].

Corollary 9. *Let $\vdash t : \mathrm{R}^d$ and v and v' be β-normal forms s.t. $t \to^* v$ and $t \to^* v'$. Then $v = v'$.*

Proof. It is not hard to show that the interpretation is injective on tuples of numerals, *i.e.*, for $v, v' : \mathrm{R}^d$ values, $[\![v]\!] = [\![v']\!]$ implies $v = v'$. The statement then follows from Props. 5 and 8. □

From now on, we will suppose that:

($\star$) all function symbols in $\mathcal{F}$ are associated by $[\![-]\!]$ with differentiable maps on real numbers.

As mentioned at the end of Sect. 2.3, this hypothesis is essentially cosmetic, in that it allows us to use actual gradients and to avoid the more technical notion of subgradient (see also Sect. 7).

Any term $x_1^{!\mathrm{R}}, \dots, x_n^{!\mathrm{R}} \vdash t : \mathrm{R}$ is denoted by an n-ary map $[\![t]\!]$ over $\mathbb{R}$. Since differentiable functions compose, then ($\star$) implies that $[\![t]\!]$ is also differentiable, if t contains only variables of type R. In the general case, one can use Prop. 6 in order to β-reduce t into a β-normal form u containing only variables of type R. Then by Prop. 8 $[\![t]\!] = [\![u]\!]$ and so $[\![t]\!]$ is differentiable. This justifies the following notation, given $x_1^{!\mathrm{R}}, \dots, x_n^{!\mathrm{R}} \vdash t : \mathrm{R}$ and a vector $\mathbf{r} \in \mathbb{R}^n$:

$$\nabla t(\mathbf{r}) := \left\langle \underline{\partial_1 [\![t]\!](\mathbf{r})}, \dots, \underline{\partial_n [\![t]\!](\mathbf{r})} \right\rangle. \tag{25}$$

Section 5 gives an efficient way of computing $\nabla t(\mathbf{r})$ based on the syntactic structure of t.

5 THE BACKPROPAGATION TRANSFORMATION

Let us fix two sets of function symbols $\mathcal{F}, \mathcal{F}'$ such that $\mathcal{F} \subseteq \mathcal{F}'$, together with partial functions $\partial i : \mathcal{F} \longrightarrow \mathcal{F}'$, for each positive integer i such that, for all $f \in \mathcal{F}$ of arity k and for all $1 \le i \le k$, $\partial_i f$ is defined and its arity is equal to k. In addition to the hypothesis ($\star$) in Sect. 4, we also suppose:

($\star\star$) $$[\![\partial_i f]\!] := \partial_i [\![f]\!]$$

For any $d \in \mathbb{N}$, Fig. 5 defines a program transformation $\overleftarrow{\mathbf{D}}_d$ from $\Lambda(\mathcal{F})$ to $\Lambda_\perp(\mathcal{F}')$, called the *reverse gradient relative to* $\mathbb{R}^d$. Given a term $\mathbf{x}^{!\mathrm{R}} \vdash t : \mathrm{R}$, Cor. 16 proves that $\overleftarrow{\mathbf{D}}_d(t)$ computes the gradient of t in at most $O(m+|G|)$ steps, where m is the cost of evaluating t to a computational graph G. Moreover, Cor. 16 is a consequence of Th. 15, proving that actually one can reduce $\overleftarrow{\mathbf{D}}_d(t)$ to a term expressing the backpropagation algorithm applied to *any* computational graph G β-equivalent to t (indeed, to a single λ-term t one can associate different computational graphs, with different sizes and with different degrees of sharing). Sect. 5.1 sets the framework needed to state and prove our results. Grammar (26) defines the notion of ground term corresponding to a computational graph and Def. 10 gives the computational graph associated with the backpropagation applied to ground terms. Prop. 12 formally proves the soundness of this algorithm. Sect. 5.2 then moves to the more general case of λ-terms, giving the soundness of our reverse gradient transformation $\overleftarrow{\mathbf{D}}_d$.

[8] More sophisticated notions of confluence modulo an extension of $\equiv$ hold for the reductions in Fig. 4, but we avoid to discuss this point here because inessential for our purposes.

$$\overleftarrow{\mathbf{D}}_d(\mathsf{R}) := \mathsf{R} \times \mathsf{R}^{\perp d} \qquad \overleftarrow{\mathbf{D}}_d(A \to B) := \overleftarrow{\mathbf{D}}_d(A) \to \overleftarrow{\mathbf{D}}_d(B) \qquad \overleftarrow{\mathbf{D}}_d(A \times B) := \overleftarrow{\mathbf{D}}_d(A) \times \overleftarrow{\mathbf{D}}_d(B)$$

(a) The action of the transformation on types.

$$\begin{aligned}
\overleftarrow{\mathbf{D}}_d(x^{!A}) &:= x^{!\overleftarrow{\mathbf{D}}_d(A)} \\
\overleftarrow{\mathbf{D}}_d(\lambda x^{!A}.t) &:= \lambda x^{!\overleftarrow{\mathbf{D}}_d(A)}.\overleftarrow{\mathbf{D}}_d(t) \\
\overleftarrow{\mathbf{D}}_d(tu) &:= \overleftarrow{\mathbf{D}}_d(t)\overleftarrow{\mathbf{D}}_d(u) \\
\overleftarrow{\mathbf{D}}_d(\langle t, u\rangle) &:= \left\langle \overleftarrow{\mathbf{D}}_d(t), \overleftarrow{\mathbf{D}}_d(u)\right\rangle \\
\overleftarrow{\mathbf{D}}_d(t[\left\langle x^{!A}, y^{!B}\right\rangle := u]) &:= \overleftarrow{\mathbf{D}}_d(t)[\left\langle x^{!\overleftarrow{\mathbf{D}}_d(A)}, y^{!\overleftarrow{\mathbf{D}}_d(B)}\right\rangle := \overleftarrow{\mathbf{D}}_d(u)] \\
\overleftarrow{\mathbf{D}}_d(t[x^{!A} := u]) &:= \overleftarrow{\mathbf{D}}_d(t)[x^{!\overleftarrow{\mathbf{D}}_d(A)} := \overleftarrow{\mathbf{D}}_d(u)] \\
\overleftarrow{\mathbf{D}}_d(\underline{r}) &:= \left\langle \underline{r}, \lambda a^{\mathsf{R}}.\underline{\mathbf{0}}\right\rangle \\
\overleftarrow{\mathbf{D}}_d(t+u) &:= \left\langle x+y, \lambda a^{\mathsf{R}}.(x^* a + y^* a)\right\rangle [\left\langle x^{!\mathsf{R}}, x^{*!\mathsf{R}^{\perp d}}\right\rangle := \overleftarrow{\mathbf{D}}_d(t)][\left\langle y^{!\mathsf{R}}, y^{*!\mathsf{R}^{\perp d}}\right\rangle := \overleftarrow{\mathbf{D}}_d(u)] \\
\overleftarrow{\mathbf{D}}_d(f(\mathbf{t})) &:= \left\langle f(\mathbf{x}),\ \lambda a^{\mathsf{R}}.\sum_{i=1}^{k} x_i^* \left(\partial_i f(\mathbf{x}) \cdot a\right)\right\rangle [\left\langle \mathbf{x}^{!\mathsf{R}}, \mathbf{x}^{*!\mathsf{R}^{\perp d}}\right\rangle := \overleftarrow{\mathbf{D}}_d(\mathbf{t})]
\end{aligned}$$

(b) The action of the transformation on terms. In the definition of $\overleftarrow{\mathbf{D}}(f(\mathbf{t}))$, the sequences $\mathbf{t}, \mathbf{x}, \mathbf{x}^*$ have all length k equal to the arity of f and the notation $[\left\langle \mathbf{x}^{!\mathsf{R}}, \mathbf{x}^{*!\mathsf{R}^{\perp d}}\right\rangle := \overleftarrow{\mathbf{D}}(\mathbf{t})]$ stands for $[\left\langle x_1^{!\mathsf{R}}, x_1^{*!\mathsf{R}^{\perp d}}\right\rangle := \overleftarrow{\mathbf{D}}(t_1)] \cdots [\left\langle x_k^{!\mathsf{R}}, x_k^{*!\mathsf{R}^{\perp d}}\right\rangle := \overleftarrow{\mathbf{D}}(t_k)]$. As mentioned in Sect. 3, the variables with superscript $*$ correspond to *backpropagators* in AD terminology [Pearlmutter and Siskind 2008].

Fig. 5. The reverse gradient $\overleftarrow{\mathbf{D}}_d$ relative to $\mathbb{R}^d$, for an arbitrary natural number d.

Let us underline that, in the last line of Fig. 5b, the index d of $\overleftarrow{\mathbf{D}}_d$ is totally independent from the arity k of the function symbol f and from the indexes $i \leq k$ of the variables x_i^* tagging the sum introduced by $\overleftarrow{\mathbf{D}}_d$. The fact that d may be arbitrary (it plays a role only in the last remark of Sect. 5.2) is a crucial feature allowing its compositionality, in contrast with the definition of $\mathbf{bp}_{\mathbf{x},a}(G)$ which has to refer to $\mathbf{x}$ containing the free variables in G (see the case $\mathbf{bp}_{\mathbf{x},a}(f(\mathbf{y}))$ in Def. 10). This being said, we henceforth omit the index d.

5.1 Backpropagation on Computational Graphs

We restrict $\Lambda(\mathcal{F})$ to terms of type R not containing higher types, deemed *ground terms*, as follows:

$$F, G ::= x^{!\mathsf{R}} \mid F[x^{!\mathsf{R}} := G] \mid f(x_1^{!\mathsf{R}}, \ldots, x_k^{!\mathsf{R}}) \mid \underline{r} \mid F + G \tag{26}$$

A term $f(G_1, \ldots, G_k)$ is considered as syntactic sugar for $f(x_1, \ldots, x_k)[x_1 := G_1] \ldots [x_k := G_k]$. Fig. 1 and 2 give examples of ground terms with the associated computational graph. Notice that the type system of Fig. 3 assigns to any ground term G a type judgment $x_1^{!\mathsf{R}}, \ldots, x_n^{!\mathsf{R}} \vdash G : \mathsf{R}$ for a suitable set of variables, so $\nabla G(\mathbf{r})$ is defined by (25) and the hypothesis ($\star$) and ($\star\star$).

We now define the transformation $\mathbf{bp}$ implementing symbolic backpropagation, as described on hypergraphs *e.g.* in [Van Iwaarden 1993, Sect. 3]. We first introduce the following notation,

evaluating some trivial sums on the fly: given two ground terms F_0, F_1,

$$F_0 \oplus F_1 := \begin{cases} F_i & \text{if } F_{i-1} = \underline{0}, \\ F_0 + F_1 & \text{otherwise} \end{cases}$$

Definition 10. Let $\Gamma = x_1^{!\mathrm{R}}, \dots, x_n^{!\mathrm{R}}$. Given a ground term of type $\Gamma \vdash G : \mathrm{R}$ and a fresh variable a^{R}, we define the term

$$\Gamma, a^{!\mathrm{R}} \vdash \mathbf{bp}_{\mathbf{x},a}(G) : \mathrm{R} \times \mathrm{R}^n$$

by induction on G, as follows. The definition is based on the inductive invariant that

$$\mathbf{bp}_{\mathbf{x},a}(G) = \langle G_0, \langle G_1, \dots, G_n \rangle\, \alpha \rangle\, \beta$$

where α and β are substitution contexts and a does not appear free in G_0 or in β.

- $\mathbf{bp}_{\mathbf{x},a}(x_i) := \left\langle x_i, \left\langle \underline{0}, \dots, a, \dots, \underline{0} \right\rangle \right\rangle$, where a appears at the i-th position in the tuple.
- $\mathbf{bp}_{\mathbf{x},a}(\underline{r}) := \left\langle \underline{r}, \left\langle \underline{0}, \dots, \underline{0} \right\rangle \right\rangle$.
- Let f be of arity k and $\mathbf{y}$ a subsequence of length k of $\mathbf{x}$. Then,

  $$\mathbf{bp}_{\mathbf{x},a}(f(\mathbf{y})) := \left\langle f(\mathbf{y}), \left\langle \underline{0}, \dots, \underline{0}, \partial_1 f(\mathbf{y}) \cdot a, \underline{0}, \dots, \underline{0}, \partial_k f(\mathbf{y}) \cdot a, \underline{0}, \dots, \underline{0} \right\rangle \right\rangle,$$

  where the non-zero terms in the tuple are at the positions determined by $\mathbf{y}$ within $\mathbf{x}$.
- Let $G = F'[z^{!\mathrm{R}} := F'']$ and suppose that (with $b^{!\mathrm{R}}$ a new variable distinct from $a^{!\mathrm{R}}$)

  $$\mathbf{bp}_{\mathbf{x},z,a}(F') = \left\langle F'_0, \left\langle F'_1, \dots, F'_n, H \right\rangle \alpha' \right\rangle \beta', \qquad \mathbf{bp}_{\mathbf{x},b}(F'') = \left\langle F''_0, \left\langle F''_1, \dots, F''_n \right\rangle \alpha'' \right\rangle \beta''.$$

  Notice that for every $i \le n$ we can suppose by renaming that the set of variables of F'_i bound by α', β' is disjoint from the set of variables of F''_i bound by α'', β''. Also, notice that F''_0 has type $\mathbf{x}^{!\mathrm{R}} \vdash F''_0 : \mathrm{R}$. So we can define:

$$\mathbf{bp}_{\mathbf{x},a}(F'[z^{!\mathrm{R}} := F'']) := \begin{cases} \left\langle F'_0, \left\langle F'_1, \dots, F'_n \right\rangle \alpha' \right\rangle \beta'[z^{!\mathrm{R}} := F''_0]\beta'' & \text{if } H = \underline{0}, \\ \left\langle F'_0, \left\langle F'_1 \oplus F''_1, \dots, F'_n \oplus F''_n \right\rangle \alpha'' [b^{!\mathrm{R}} := H]\alpha' \right\rangle \beta'[z^{!\mathrm{R}} := F''_0]\beta'' & \text{otherwise.} \end{cases}$$

- Let $G = F' + F''$ and suppose that

  $$\mathbf{bp}_{\mathbf{x},a}(F') = \left\langle F'_0, \left\langle F'_1, \dots, F'_n \right\rangle \alpha' \right\rangle \beta', \qquad \mathbf{bp}_{\mathbf{x},a}(F'') = \left\langle F''_0, \left\langle F''_1, \dots, F''_n \right\rangle \alpha'' \right\rangle \beta'',$$

  then, $\mathbf{bp}_{\mathbf{x},a}(F' + F'') := \left\langle F'_0 + F''_0, \left\langle F'_1 \oplus F''_1, \dots, F'_n \oplus F''_n \right\rangle \alpha' \alpha'' \right\rangle \beta' \beta''$.

Lemma 11. *Let G be a ground term with $\mathrm{fv}(G) = \mathbf{x}$, then $\left|\mathbf{bp}_{\mathbf{x},a}(G)\right| = O(|G|)$.*

Proposition 12 (Soundness of **bp**). *Let G be a ground term whose free variables are given by a sequence $\mathbf{x}$ of length n. Then for every $\mathbf{r} \in \mathbb{R}^n$, we have:*

$$\mathbf{bp}_{\mathbf{x},a}(G)[a := \underline{1}][\mathbf{x} := \underline{\mathbf{r}}] \xrightarrow{(10)^{\mathrm{n}}(11)(13)(15)} {}^{O(|G|)}\equiv \left\langle \underline{[\![G]\!]\,(\mathbf{r})}, \nabla G(\mathbf{r}) \right\rangle.$$

Proof. Let $R = \{(10)^{\mathrm{n}}(11)(13)(15)\}$. By induction on G we prove that, supposing $\mathbf{bp}_{\mathbf{x},a}(G) = \langle G_0, \langle G_1, \dots, G_n \rangle\, \alpha \rangle\, \beta$, for every vector of real numbers $\mathbf{r}$, we have: (i) $G_0\beta[\mathbf{x} := \underline{\mathbf{r}}] \xrightarrow{R}{}^{*}\equiv \underline{[\![G]\!]\,(\mathbf{r})}$; (ii) for all $1 \le i \le n$, for all real number q, $G_i\alpha\beta[a := \underline{q}][\mathbf{x} := \underline{\mathbf{r}}] \xrightarrow{R}{}^{*}\equiv \underline{\partial_{x_i}\,[\![G]\!]\,(\mathbf{r}) \cdot q}$. From (i) and (ii): $\mathbf{bp}_{\mathbf{x},a}(G)[a := \underline{1}][\mathbf{x} := \underline{r}] \equiv \left\langle G_0\beta[\mathbf{x} := \underline{\mathbf{r}}], \left\langle G_1\alpha\beta[a := \underline{1}][\mathbf{x} := \underline{\mathbf{r}}], \dots, G_n\alpha\beta[a := \underline{1}][\mathbf{x} := \underline{\mathbf{r}}] \right\rangle \right\rangle \xrightarrow{R}{}^{*}\equiv \left\langle \underline{[\![G]\!]\,(\mathbf{r})}, \underline{\nabla_{\mathbf{r}}\,[\![G]\!]} \right\rangle$. Prop. 4 allows to postpone all structural equivalences at the end. From Lem. 7 and $\left|\mathbf{bp}_{\mathbf{x},a}(G)\right| = O(|G|)$ (Lem. 11) the length of the reduction is $O(|G|)$.

We give only the proof of (ii) for $G = F'[z := F'']$, the other cases being similar or trivial. Using the notations from Def. 10 we have that the i-th component of the gradient tuple of $\mathbf{bp}_{\mathbf{x},a}(G)$ together with the associated substitutions is equal to (we suppose $F'_i \oplus F''_i = F'_i + F''_i$, the other cases being simpler):

$$
\begin{aligned}
&(F'_i + F''_i)\alpha''[b := H]\alpha'\beta'[z := F''_0]\beta''[a := \underline{q}][\mathbf{x} := \underline{\mathbf{r}}]\\
&\equiv (F'_i + F''_i)\alpha''[b := H]\alpha'\beta'\beta''[a := \underline{q}][\mathbf{x} := \underline{\mathbf{r}}][z := F''_0\beta''[\mathbf{x} := \underline{\mathbf{r}}]]\\
&\xrightarrow{R}{}^{*} (F'_i + F''_i)\alpha''[b := H]\alpha'\beta'\beta''[a := \underline{q}][\mathbf{x} := \underline{\mathbf{r}}][z := \underline{[\![F'']\!](\mathbf{r})}] && \text{by (i)}\\
&\equiv (F'_i + F''_i)\alpha''\alpha'\beta'\beta''[a := \underline{q}][\mathbf{x} := \underline{\mathbf{r}}][z := \underline{[\![F'']\!](\mathbf{r})}][b := H\alpha'\beta'\beta''[a := \underline{q}][\mathbf{x} := \underline{\mathbf{r}}][z := \underline{[\![F'']\!](\mathbf{r})}]]\\
&\xrightarrow{R}{}^{O(|F'|)} (F'_i + F''_i)\alpha''\alpha'\beta'\beta''[a := \underline{q}][\mathbf{x} := \underline{\mathbf{r}}][z := \underline{[\![F'']\!](\mathbf{r})}][b := \underline{\partial_z [\![F']\!](\mathbf{r}) \cdot q}] && \text{by IH}\\
&\equiv F'_i\alpha'\beta'[a := \underline{q}][\mathbf{x} := \underline{\mathbf{r}}][z := \underline{[\![F'']\!](\mathbf{r})}] + F''_i\alpha''\beta''[b := \underline{\partial_z [\![F']\!](\mathbf{r}) \cdot q}][\mathbf{x} := \underline{\mathbf{r}}]\\
&\xrightarrow{R}{}^{*} \underline{\partial_{x_i} [\![F']\!](\mathbf{r}) \cdot q} + \underline{\partial_{x_i} [\![F'']\!](\mathbf{r}) \cdot (\partial_z [\![F']\!](\mathbf{r}) \cdot q)} && \text{by IH}\\
&\xrightarrow{(13)} \underline{\left(\partial_{x_i} [\![F']\!](\mathbf{r}) + \partial_z [\![F']\!](\mathbf{r}) \cdot \partial_{x_i} [\![F'']\!](\mathbf{r})\right) \cdot q = \partial_{x_i} [\![G]\!](\mathbf{r}) \cdot q}
\end{aligned}
$$

Notice that in order to move to the last line we use the associativity, commutativity and distributivity of + and · over real numbers, not on the corresponding syntactic symbols. In the base cases (variables, functional symbols and numerals), one performs linear substitutions (10) as well as garbage collection (11). In the case of $\mathbf{bp}_{\mathbf{x},a}(f(x_{i_1}, \ldots, x_{i_k}))$ one instance of rule (15) is needed. □

Notice that the result of the evaluation of $\mathbf{bp}_{\mathbf{x},a}(G)[a := \underline{1}][\mathbf{x} := \underline{\mathbf{r}}]$ is independent from the chosen reduction sequence by Corollary 9.

5.2 Backpropagation on Higher-Order Programs

Let us now consider the soundness of our transformation $\overleftarrow{\mathbf{D}}(t)$ applied to any λ-term of type $x_1^{!\mathrm{R}}, \ldots, x_n^{!\mathrm{R}} \vdash t : \mathrm{R}$. The proof uses two essential ingredients: first, we remark that the transformation $\overleftarrow{\mathbf{D}}(t)$ commutes with any reduction step (Lem. 13); second, we prove that $\overleftarrow{\mathbf{D}}$ applied to a computational graph G encodes actually all the relevant information of $\mathbf{bp}_{\mathbf{x},a}(G)$ (Lem. 14). Notice that Lem. 14 introduces fresh variables x'_i $(i \leq n)$ annotated $!\mathrm{R}^{\perp}$, tagging the different components of the gradient of G in a sum. We can then conclude with Th. 15 and its Cor. 16.

Lemma 13. *Let t be a term of $\Lambda(\mathcal{F})$. Then:*

1. *if $t \equiv t'$, then $\overleftarrow{\mathbf{D}}(t) \equiv \overleftarrow{\mathbf{D}}(t')$,*
2. *if $t \xrightarrow{\iota} t'$, for ι any reduction step in Fig. 4, then $\overleftarrow{\mathbf{D}}(t) \xrightarrow{X}{}^{O(1)} \overleftarrow{\mathbf{D}}(t')$, where $X = \{\iota\}$ for any ι but (13), (15), in these latter cases $X = \{\iota, (8), (10), (11)\}$.*

Lemma 14. *Let G be a ground term with $\mathrm{fv}(G) = \mathbf{x} = x_1^{!\mathrm{R}}, \ldots, x_n^{!\mathrm{R}}$ and suppose that $\mathbf{bp}_{\mathbf{x},a}(G) = \langle G_0, \langle G_1, \ldots, G_n\rangle\, \alpha\rangle\, \beta$. Then, there exists $J \subseteq \{1, \ldots, n\}$ such that $i \notin J$ implies $G_i = \underline{0}$ and such that*

$$\overleftarrow{\mathbf{D}}(G)[\mathbf{x}^{!\overleftarrow{\mathbf{D}}(\mathrm{R})} := \left\langle \mathbf{x}^{!\mathrm{R}}, \lambda a^{\mathrm{R}}.\mathbf{x}^{*!\mathrm{R}^{\perp}} a \right\rangle] \to^{O(|G|)} \equiv \left\langle G_0, \lambda a.(\sum_{j\in J} x_j^* G_j)\alpha \right\rangle \beta.$$

Proof. By induction on G. We only consider $G = F[z := F']$, the other cases being simpler. Let $\mathbf{bp}_{\mathbf{x},z,a}(F) = \langle F_0, \langle F_1, \ldots, F_n, H\rangle\, \alpha\rangle\, \beta$ and $\mathbf{bp}_{\mathbf{x},b}(F') = \left\langle F'_0, \left\langle F'_1, \ldots, F'_n\right\rangle \alpha'\right\rangle \beta'$. Let us also consider

$H \neq \underline{0}$ (the other case being simpler). The term $\overleftarrow{\mathbf{D}}(G)[\mathbf{x} := \langle \mathbf{x}, \lambda a.\mathbf{x}^* a\rangle]$ is structural equivalent to:

$$(\overleftarrow{\mathbf{D}}(F)[\mathbf{x} := \langle \mathbf{x}, \lambda a.\mathbf{x}^* a\rangle])[z := (\overleftarrow{\mathbf{D}}(F')[\mathbf{x} := \langle \mathbf{x}, \lambda b.\mathbf{x}^* b\rangle])]$$

$$\to^{O(|F'|)} \equiv (\overleftarrow{\mathbf{D}}(F)[\mathbf{x} := \langle \mathbf{x}, \lambda a.\mathbf{x}^* a\rangle])[z := \left\langle F'_0, \lambda b.(\sum_{j\in J'} x_j^* F'_j)\alpha' \right\rangle \beta'] \qquad \text{by IH}$$

$$\equiv (\overleftarrow{\mathbf{D}}(F)[\mathbf{x} := \langle \mathbf{x}, \lambda a.\mathbf{x}^* a\rangle])[z := \left\langle F'_0, \lambda b.(\sum_{j\in J'} x_j^* F'_j)\alpha' \right\rangle]\beta'$$

$$\xrightarrow{(17)}\xrightarrow{(9)} (\overleftarrow{\mathbf{D}}(F)[\mathbf{x} := \langle \mathbf{x}, \lambda a.\mathbf{x}^* a\rangle][z := \langle z, z^*\rangle])[z := F'_0][z^* := \lambda b.(\sum_{j\in J'} x_j^* F'_j)\alpha']\beta'$$

$$\xrightarrow{(16)} (\overleftarrow{\mathbf{D}}(F)[\mathbf{x} := \langle \mathbf{x}, \lambda a.\mathbf{x}^* a\rangle][z := \langle z, \lambda a.z^* a\rangle])[z := F'_0][z^* := \lambda b.(\sum_{j\in J'} x_j^* F'_j)\alpha']\beta'$$

$$\to^{O(|F|)} \equiv \left\langle F_0, \lambda a.(\sum_{j\in J} x_j^* F_j + z^* H)\alpha \right\rangle \beta[z := F'_0][z^* := \lambda b.(\sum_{j\in J'} x_j^* F'_j)\alpha']\beta' \qquad \text{by IH}$$

$$\xrightarrow{(10)}\xrightarrow{(11)} \left\langle F_0, \lambda a.(\sum_{j\in J} x_j^* F_j + (\lambda b.(\sum_{j\in J'} x_j^* F'_j)\alpha')H)\alpha \right\rangle \beta[z := F'_0]\beta'$$

$$\xrightarrow{(8)} \left\langle F_0, \lambda a.(\sum_{j\in J} x_j^* F_j + (\sum_{j\in J'} x_j^* F'_j)\alpha'[b := H])\alpha \right\rangle \beta[z := F'_0]\beta'$$

$$\xrightarrow{(18)}{}^{\#J\cap\#J'} \left\langle F_0, \lambda a.(\sum_{j\in J\cup J'} x_j^*(F_j \oplus F'_j))\alpha'[b := H]\alpha \right\rangle \beta[z := F'_0]\beta'$$

□

By composing the reductions in Lemma 13 and Lemma 14, we get:

Theorem 15. *Let t be a term of $\Lambda(\mathcal{F})$ of type R with $\mathrm{fv}(t) = \mathbf{x} = x_1^{!\mathsf{R}}, \dots, x_n^{!\mathsf{R}}$. For any ground term G such that $t\,(\xrightarrow{\beta} \cup \equiv)^* G$ in m β-steps, we have, for a suitable $J \subseteq \{1, \dots, n\}$:*

$$\overleftarrow{\mathbf{D}}(t)[\mathbf{x}^{!\overleftarrow{\mathbf{D}}(\mathsf{R})} := \left\langle \mathbf{x}^{!\mathsf{R}}, \lambda a^{\mathsf{R}}.\mathbf{x}^{*!\mathsf{R}^\perp} a \right\rangle] \to^{O(m+|G|)} \equiv \left\langle G_0, \lambda a.(\sum_{j\in J} x_j^* G_j)\alpha \right\rangle \beta$$

where $\mathbf{bp}_{\mathbf{x},a}(G) = \langle G_0, \langle G_1, \dots, G_n\rangle\, \alpha\rangle\, \beta$ and $\forall i \notin J, G_i = \underline{0}$.

Corollary 16. *Let t be a term of $\Lambda(\mathcal{F})$ of type $\mathbf{x}^{!\mathsf{R}} \vdash t : \mathsf{R}$, with $\mathbf{x} = x_1^{!\mathsf{R}}, \dots, x_n^{!\mathsf{R}}$ the free variables of t. Let m be the number of β-steps needed to reduce t to a ground term G. For any vector $\mathbf{r} \in \mathbb{R}^n$, let $\nabla t(\mathbf{r}) = \left\langle \underline{g_1}, \dots, \underline{g_n} \right\rangle$. Then, for a suitable $J \subseteq \{1, \dots, n\}$, with $i \notin J$, $g_i = 0$, we have:*

$$z^*\underline{1}[\langle z, z^*\rangle := \overleftarrow{\mathbf{D}}(t)[\mathbf{x} := \langle \mathbf{x}, \lambda a.\mathbf{x}^* a\rangle]][\mathbf{x} := \underline{\mathbf{r}}] \to^{O(m+|G|)} \equiv \sum_{j\in J} x_j^* \underline{g_j}.$$

One can go further and obtain the tuple of numerals $\nabla t(\mathbf{r})$ from $\sum_{j\in J} x_j^* \underline{g_j}$ just by: (i) considering $\overleftarrow{\mathbf{D}}_d(t)$ with $d = n$ (here is the only point where we require a specific choice of d); (ii) replacing each x_i^* ($i \leq n$), morally of type $\mathsf{R} \multimap \mathsf{R}^n$, with the corresponding injection $\lambda a.\, \langle 0 \dots, 0, a, 0, \dots, 0\rangle$; (iii) adding all tuples resulting from the reductions (8),(10),(11): $\sum_{j\in J} \left\langle \underline{0}, \dots, \underline{0}, \underline{g_j}, \underline{0} \dots, \underline{0} \right\rangle \to^*$

$\left\langle \underline{g}_1, \ldots, \underline{g}_n \right\rangle$. However, this last reduction is quadratic in n, as it performs $\#J = O(n)$ sums of vectors of size n without considering that these latter are null but on one coordinate. It would then be preferable to add an *ad hoc* read-back operation allowing to decode the tuple $\nabla t(\mathbf{r})$ out of the tagged sum $\sum_{j\in J} x_j^* \underline{g_j}$ more parsimoniously.

Notice that the proof of Cor. 16 considers a specific reduction sequence for getting $\nabla t(\mathbf{r})$. However, Cor. 9 guarantees that the result is independent from the chosen sequence.[9] This suggests considering more efficient strategies than that of Cor. 16, or even more modular by decomposing the computation along the different components of t. We discuss this last point in the next Section.

6 EXAMPLES

6.1 Derivative of a Dynamically Generated Polynomial

Let $\mathsf{Nat} := (\mathsf{R} \to \mathsf{R}) \to \mathsf{R} \to \mathsf{R}$ be the type of Church natural numbers and consider

$$G := w \cdot y + x \qquad\qquad t := \lambda n^{!\mathsf{Nat}}.\lambda x.n(\lambda y.G)x.$$

We have $x^{!\mathsf{R}}, y^{!\mathsf{R}}, w^{!\mathsf{R}} \vdash G : \mathsf{R}$ and $w^{!\mathsf{R}} \vdash t : \mathsf{Nat} \to \mathsf{R} \to \mathsf{R}$. If $\underline{n}$ encodes $n \in \mathbb{N}$, the term $t\,\underline{n}$ dynamically generates the function $\lambda x.f_n(w,x)$ with $f_n(w,x) := (w^n + w^{n-1} + \cdots + w + 1) \cdot x$ (we take some liberty in simplifying arithmetic expressions for the sake of readability). Notice that the free variable x of $\lambda y.G$, a term to be iterated at will, is captured later in t. We have

$$\overleftarrow{\mathbf{D}}(\lambda y.G) \to^* \lambda\langle y, y^*\rangle.\, \langle G, \lambda a.w^*(y \cdot a) + y^*(w \cdot a) + x^* a\rangle =: G',$$

where we used the shorthand $\lambda\langle y, y^*\rangle.u := \lambda p.u[\langle y, y^*\rangle := p]$. Hence, when *e.g.* $n = 2$,

$$\begin{aligned}\overleftarrow{\mathbf{D}}(t\,\underline{2}) = (\lambda n.\lambda x.n\overleftarrow{\mathbf{D}}(\lambda y.G)x)\,\underline{2} &\to^* (\lambda n.\lambda x.n\,G'\,x)\,\underline{2} \to \lambda x.\underline{2}\,G'\,x \to \lambda x.G'(G'x)\\ &\to \lambda\langle x, x^*\rangle.G'(G'\,\langle x, x^*\rangle) \to^* \lambda\langle x, x^*\rangle.G'\,\langle wx + x, H\rangle\,,\end{aligned}$$

where $H := \lambda a.w^*(x \cdot a) + x^*(w \cdot a) + x^* a$. The computation continues with

$$\begin{aligned}&\to^* \lambda\langle x, x^*\rangle.\,\left\langle (w^2 + w + 1)x, \lambda a.w^*((wx + x) \cdot a) + H(w \cdot a) + x^* a\right\rangle\\ &\to^* \lambda\langle x, x^*\rangle.\,\left\langle (w^2 + w + 1)x, \lambda a.w^*((wx + x) \cdot a) + w^*(wx \cdot a) + x^*(w^2 \cdot a) + x^*(w \cdot a) + x^* a\right\rangle\\ &\to^* \lambda\langle x, x^*\rangle.\,\left\langle (w^2 + w + 1)x, \lambda a.w^*((2wx + x) \cdot a) + x^*((w^2 + w + 1) \cdot a)\right\rangle\end{aligned}$$

We see that the argument of w^* (resp. x^*) is the derivative of f_2 with respect to w (resp. x). This shows how locally free variables are handled correctly. Also observe that G', which is trivial here but could in principle be a very complex function, may be pre-computed independently of n.

6.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) are meant to process inputs that are arbitrary sequences of vectors in $\mathbb{R}^d$. They are heavily used for natural language processing: sequences could for instance stand for a word or a sentence, where each vector is a representation of a letter. Such networks iterate over the input to recursively build a desired output. One example is *encoding recurrent neural networks*, which are used to encode a sequence of vectors $x_i \in \mathbb{R}^d$ into an output vector $h \in \mathbb{R}^m$. For instance, in *sentiment analysis* [dos Santos and Gatti 2014; Glorot et al. 2011; Severyn and Moschitti 2015], it is used to predict if the expressed opinion in a sentence is positive, negative or neutral.

Given a sequence $x_1, \ldots, x_n \in \mathbb{R}^d$, the encoding RNN produces intermediate outputs $h_1, \ldots, h_n \in \mathbb{R}^m$ recursively, by applying a single layer $L : \mathbb{R}^d \times \mathbb{R}^m \to \mathbb{R}^m$ to both the last output value h_i and

[9] This is in fact the case if one admits the injections $\lambda a.\,\langle 0 \ldots, 0, a, 0, \ldots, 0\rangle$ of type $\mathsf{R}^\perp$, as discussed in the previous note.

the next input vector x_{i+1}:

$$h_{i+1} = L(x_{i+1}, h_i) \qquad L(x,h) = \sigma(E \cdot x + R \cdot h)$$

where σ is the sigmoid activation function, E is a $d \times m$-matrix called the *token embedding matrix*, and R is a $m \times m$ matrix whose coefficients are called *the recurrent weights*. h_0 can be initialized to $\mathbf{0}$. In practice, a RNN ends with a loss function that we are looking to minimize (and which models the problem we want to solve). To keep the example simple, we will not consider it. For the same reason, we suppose that $d = m = 1$, the general encoding being a direct generalization.

Lists. We represent lists in our language using the Church encoding, which defines a list by its right fold function. Given A and X be types, the type of lists $\mathsf{List}(A,X)$ is defined as follows:

$$\mathsf{List}(A,X) := (A \to X \to X) \to (X \to X)$$

Given $a_1 : A, \dots, a_n : A$, we define the list $[a_1; \dots; a_n]_X$ of type $\mathsf{List}(A,X)$ as:

$$[a_1; \dots; a_n]_X := \lambda f^{!(A \to X \to X)}.\lambda x^{!X}.f\,a_1(f\,a_2 \dots (f\,a_n x)\dots)$$

In what follows we will omit the X annotation if it is clear from context or if it does not matter. Finally, it can be noted that the reverse gradient of the list datatype is given by:

$$\overleftarrow{\mathbf{D}}(\mathsf{List}(A,X)) = \mathsf{List}(\overleftarrow{\mathbf{D}}(A), \overleftarrow{\mathbf{D}}(X)) \qquad \overleftarrow{\mathbf{D}}([a_1; \dots; a_n]_X) = [\overleftarrow{\mathbf{D}}(a_1), \dots, \overleftarrow{\mathbf{D}}(a_n)]_{\overleftarrow{\mathbf{D}}(X)}$$

Encoding a RNN. Let σ be the function symbol corresponding to the sigmoid function $\mathbb{R} \to \mathbb{R}$. In dimension 1, the token embedding matrix and the recurrent matrix may be represented as two terms $\lambda x^{!\mathsf{R}}.(\epsilon \cdot x)$ and $\lambda h^{!\mathsf{R}}.(\rho \cdot h)$ respectively, where $\epsilon^{!\mathsf{R}}$ and $\rho^{!\mathsf{R}}$ are two variables. The recurring layer L and the recurring network N is then defined and typed as follows:

$$L := \lambda x^{!\mathsf{R}}.\lambda h^{!\mathsf{R}}.\sigma(\epsilon \cdot x + \rho \cdot h) \qquad \epsilon^{!\mathsf{R}}, \rho^{!\mathsf{R}} \vdash L : \mathsf{R} \to \mathsf{R} \to \mathsf{R}$$

$$N := \lambda l^{!List(\mathsf{R},\mathsf{R})}.l L \underline{0} \qquad \epsilon^{!\mathsf{R}}, \rho^{!\mathsf{R}} \vdash N : \mathsf{List}(\mathsf{R},\mathsf{R}) \to \mathsf{R}$$

The free variables ϵ and ρ are the parameters that we wish to learn via gradient descent.

Backpropagation. For RNNs, the gradient is usually computed using a technique called *backpropagation through time* [Pearlmutter 1995; Rumelhart et al. 1987]. The method consists in unfolding the RNN (by applying it to an input sequence) and then applying the usual backpropagation over plain vanilla feedforward neural networks. We will now apply our reverse gradient transformation to an example of RNN, and show that backpropagation through time is naturally implemented by the reduction strategy of Theorem 15.

Given $l = [\underline{a_1}; \dots; \underline{a_n}]_{\mathsf{R}}$, we compute the gradient of Nl with respect to ϵ and ρ. The following proposition exposes the recursive equations expressing the gradient computed thanks to our transformation. They are similar to the equations of backpropagation through time.

Proposition 17. *We have* $\nabla(Nl)(e,r) = \left\langle g^n_\epsilon, g^n_\rho \right\rangle$ *where* g^n_ϵ *and* g^n_ρ *are given by the following recurrent equations:*

$$g^0_\epsilon = \underline{0} \qquad g^{i+1}_\epsilon = \sigma'_{i+1} \cdot (\underline{a_{i+1}} + \underline{r} \cdot g^i_\epsilon) \qquad \sigma'_{i+1} := \partial_1\sigma(\underline{e} \cdot \underline{a_{i+1}} + \underline{r} \cdot u_i)$$

$$g^0_\rho = \underline{0} \qquad g^{i+1}_\rho = \sigma'_{i+1} \cdot (u_{i+1} + \underline{r} \cdot g^i_\rho)$$

$$u_0 = \underline{0} \qquad u_{i+1} = \sigma(\underline{e} \cdot \underline{a_{i+1}} + \underline{r} \cdot u_i)$$

Proof. By Corollary 16, this amounts to computing $D = \epsilon^* g_\epsilon + \rho^* g_\rho$ such that

$$z^* \underline{1}[\langle z, z^* \rangle := \overleftarrow{\mathbf{D}}(Nl)[\epsilon := \langle \epsilon, \lambda a.\epsilon^* a \rangle][\rho := \langle \rho, \lambda a.\rho^* a \rangle]][\epsilon := \underline{e}][\rho := \underline{r}] \to^* D$$

This is done by induction over the size of the list l. □

The strategy from Theorem 15 first computes $\overleftarrow{\mathbf{D}}(Nl)$, then reduces it to $\overleftarrow{\mathbf{D}}(G)$ such that $Nl \to^* G$, with G a ground term, *i.e.*, only containing subterms of ground type. Here:

$$G = F(\underline{a_1}, F(\underline{a_2}, \dots, F(a_n, \underline{0}) \dots))$$

where $L = \lambda x.\lambda h.F$ with $F = \epsilon \cdot x + \rho \cdot h$. Notice that G is exactly the unfolding of the RNN, which should convince the reader that this strategy implements the backpropagation through time.

We now turn our attention to the effectiveness of the reduction strategy. During the reduction of Nl to G, the λ-abstractions of L must be eliminated. Each of these λ is applied exactly to n different arguments, requiring L to be duplicated n times. In other words, we satisfy the following reduction:

$$Nl \to^* L\underline{a_1}(L\underline{a_2} \dots (L\underline{a_n}\underline{0}) \dots) \to^* G$$

By point 2 of Lemma 13, the exact same reasoning may be applied to $\overleftarrow{\mathbf{D}}(Nl)$, in which $\overleftarrow{\mathbf{D}}(L)$ must be duplicated n times:

$$\overleftarrow{\mathbf{D}}(Nl) \to^* \overleftarrow{\mathbf{D}}(L)\overleftarrow{\mathbf{D}}(\underline{a_1})(\overleftarrow{\mathbf{D}}(L)\overleftarrow{\mathbf{D}}(\underline{a_2}) \dots (\overleftarrow{\mathbf{D}}(L)\overleftarrow{\mathbf{D}}(\underline{a_n})\overleftarrow{\mathbf{D}}(\underline{0})) \dots) \to^* \overleftarrow{\mathbf{D}}(G)$$

A simple observation shows that $\overleftarrow{\mathbf{D}}(L)$ is not in β-normal form:

$$\overleftarrow{\mathbf{D}}(L) = \lambda x.\lambda h.\, \langle \sigma(\mu), \lambda a.\mu^*(\partial_1\sigma(\mu) \cdot a)\rangle\, [\langle \mu, \mu^* \rangle := \overleftarrow{\mathbf{D}}(\epsilon \cdot x + \rho \cdot h)]$$

$\overleftarrow{\mathbf{D}}(\epsilon \cdot x + \rho \cdot h)$ being a pair, we have $\overleftarrow{\mathbf{D}}(L) \xrightarrow{\beta}{}^{k} u$ for some β-normal M. Moreover, it can be shown that when the dimensions d and m are > 1, $k = O((d + m) \times m)$. In our case, we see that each component of the sum in $\overleftarrow{\mathbf{D}}(\epsilon \cdot x + \rho \cdot h)$ will be responsible for at least one substitution. Finally, since $\overleftarrow{\mathbf{D}}(L)$ is duplicated n times in the original backpropagation through time strategy, by using the strategy where $\overleftarrow{\mathbf{D}}(L)$ is reduced to M before being substituted, we obtain a gain of $O((d+m)mn)$ reduction steps. This is significant as the number of learning parameters (here $(d + m)m$) can be very large in typical neural networks. For instance, a recent model called GPT-2 [Radford et al. 2019] has 1.5 billion parameters.

7 DISCUSSION AND PERSPECTIVES

On Expressiveness. The simply-typed λ-calculus is certainly too restrictive as a programming language, but it does present the main obstacle in defining higher-order backpropagation, namely the native use of higher-order without necessarily going through computational graphs. It also has a minimum of expressiveness for interesting examples to exist, such as inductive types with very basic operations on them (`map`, `fold`), as shown in Sect. 6. In fact, our results apply seamlessly to any *total* language with set-theoretic semantics (*e.g.* Gödel's System T with arbitrary inductive types), with no need of additional technical ideas. This is already quite broad: no state-of-the-art deep learning architecture we are aware of (convolutional NNs, RNNs, Tree-RNNs, attention-based NNs, transformers…) requires going beyond System T in order to be expressed.

We wish to stress that our current proof *does* support, unchanged, non-differentiable functions like ReLU. As mentioned at the end of Sect. 2.3, the differentiability hypothesis ($\star$) (end of Sect. 4) is essentially cosmetic, in that it allows us to use actual gradients and to avoid the more technical notion of subgradient. In the absence of ($\star$), Corollary 16 holds for all vectors in the domain of definition of Eq. 25, *i.e.*, wherever the gradient makes sense.

Still in relation with partiality, we argue that our framework can accommodate full recursion (like [Wang et al. 2019]), by adding the definition $\overleftarrow{\mathbf{D}}(\mathsf{Y}_A) := \mathsf{Y}_{\overleftarrow{\mathbf{D}}(A)}$ to Fig. 5, where $\mathsf{Y}_A : (A \to A) \to A$ is the fixpoint operator. As for denotational semantics, one has to consider the cartesian closed category of pointed complete partial order sets and Scott-continuous functions. The type R is

interpreted as the flat domain having a bottom element (representing non-termination) and all real numbers as maximal elements. Corollary 16 then holds for all vectors on which the program converges and is differentiable. However, this still falls short of being a full programming language like PCF because it lacks conditionals (on R). The issues related to the presence of branchings are mentioned in [Plotkin 2018], where a proposal is suggested for first-order languages.

On Complexity. In computational graphs, complexity analysis assumes that computing the value of a single node from its local inputs has unitary cost. In terms of low-level complexity models (*e.g.* random access machines), this rests on the assumption that *searching for the next node to evaluate is a constant-time operation*, which is fair because we may suppose the nodes to be linearly ordered compatibly with the dependencies of the graph.

Our analysis shows that, as soon as the programming language is suitably fine-grained, it is consistent to attribute a unitary cost to a single evaluation step (*i.e.*, a rewriting rule from Fig. 4). However, in general programming languages, it is unclear whether the search for the next redex has constant cost. Albeit recent work [Accattoli and Barras 2017, Corollary 9.2] shows that this is the case in call-by-name evaluation of closed programs, a detailed analysis for our language is currently missing and we defer it to future work.

On Higher-Type Derivatives. A major endeavor relating functional primitives with differential operators is differential linear logic [Ehrhard 2018] and its associated differential λ-calculus [Ehrhard and Regnier 2003]. Indeed, the initial motivation of our work was to express the backpropagation algorithm in the differential λ-calculus. Although this is possible, we realized that it required discarding the most important programming primitive of the differential λ-calculus, namely the derivative operator D on higher-order types. This is probably a good place to point out a crucial feature of our program transformation $\overleftarrow{\mathrm{D}}$. Take a term $\lambda x.t(ux)$ of type $\mathrm{R} \to \mathrm{R}$ and such that $\vdash t : A \to \mathrm{R}$ and $\vdash u : \mathrm{R} \to A$. We have two different ways of computing the derivative of $t(ux)$ with respect to x^{R}: either by our transformation $\overleftarrow{\mathrm{D}}(\lambda x.t(ux))$ or by Ehrhard's derivative operator $\mathrm{D}(\lambda x.t(ux))$. Both ways are purely functional, but our operator decomposes as $\overleftarrow{\mathrm{D}}(\lambda x.t(ux)) =_\beta \lambda \langle x, a\rangle .\overleftarrow{\mathrm{D}}(t)(\overleftarrow{\mathrm{D}}(u)\langle x, a\rangle)$, while Ehrhard's follows the chain rule (see Sect. 2.3), giving $\mathrm{D}(\lambda x.t(ux)) =_\beta \lambda \langle x, a\rangle .\mathrm{D}(t)\langle ux, \mathrm{D}(u)\langle x, a\rangle\rangle$.[10] It is clear that this latter expression is inefficient with respect to backpropagation because of the duplication of the term u (see also Sect. 2).

So our $\overleftarrow{\mathrm{D}}$ and the operator D of the differential λ-calculus are extensionally different. This is immediately seen on "pure" λ-terms (with no function symbols): when A is higher order, there are pure terms $t : A \to \mathrm{R}$ with non-trivial derivative, which is *always* computed by $\mathrm{D}(t)$, whereas $\overleftarrow{\mathrm{D}}(t) = t$ (modulo a type change), which cannot compute the derivative.[11] Indeed, observe that our soundness statements (Theorem 15, Corollary 16) only hold for terms of ground type. At present, we seem to have no use for the extra generality provided by the differential λ-calculus, but it is a natural and intriguing question to understand whether the ability to compute derivatives at higher types (rather than just over R) can play a role in developing new machine learning models.

ACKNOWLEDGMENTS

We would like to thank T. Ehrhard, C. Fouqueré, M. Gaboardi, B.A. Pearlmutter, Y. Regis-Gianas, J.M. Siskind, C. Tasson and the anonymous reviewers for useful comments and discussions.

[10] Here we use some syntactic sugar in order to avoid notational bureaucracy. For example, $\lambda \langle x, a\rangle \ldots$ stands for $\lambda y. \ldots [\langle x, a\rangle := y]$ in our language. Also, the differential λ-calculus of [Ehrhard and Regnier 2003] does not have pairs and so $\lambda \langle x, a\rangle$ is curried into $\lambda x.\lambda a$.

[11] This does not contradict Corollary 16 because, when A is of order zero (*i.e.*, $A = \mathrm{R}^n$), any pure t must be a projection and the above equality is correct.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of OSDI*. USENIX Association, 265–283.

Beniamino Accattoli. 2012. An Abstract Factorization Theorem for Explicit Substitutions. In *Proceedings of RTA (LIPIcs)*, Vol. 15. 6–21.

Beniamino Accattoli. 2018. Proof Nets and the Linear Substitution Calculus. In *Proceedings of ICTAC (Lecture Notes in Computer Science)*, Vol. 11187. Springer, 37–61.

Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. 2014. Distilling Abstract Machines. In *Proceedings of ICFP*. ACM, 363–376.

Beniamino Accattoli and Bruno Barras. 2017. Environments and the complexity of abstract machines. In *In Proceedings of PPDP*. ACM, 4–16.

Atılım Güneş Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2017. Automatic Differentiation in Machine Learning: a Survey. *Journal of Machine Learning Research* 18 (2017), 153:1–153:43.

George Cybenko. 1989. Approximation by superpositions of a sigmoidal function. *MCSS* 2, 4 (1989), 303–314.

Olivier Danvy and Mayer Goldberg. 2005. There and Back Again. *Fundam. Inform.* 66, 4 (2005), 397–413.

Cicero dos Santos and Maira Gatti. 2014. Deep convolutional neural networks for sentiment analysis of short texts. In *Proceedings of COLING: Technical Papers*. 69–78.

Thomas Ehrhard. 2018. An introduction to differential linear logic: proof-nets, models and antiderivatives. *Mathematical Structures in Computer Science* 28, 7 (2018), 995–1060.

Thomas Ehrhard and Giulio Guerrieri. 2016. The Bang Calculus: an untyped lambda-calculus generalizing call-by-name and call-by-value. In *Proceedings PPDP*. ACM, 174–187.

Thomas Ehrhard and Laurent Regnier. 2003. The differential lambda-calculus. *Theor. Comput. Sci.* 309, 1-3 (2003), 1–41.

Conal Elliott. 2018. The simple essence of automatic differentiation. *PACMPL* 2, ICFP (2018), 70:1–70:29.

Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50, 1 (Jan. 1987), 1–102.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Domain adaptation for large-scale sentiment classification: A deep learning approach. In *Proceedings of ICML*. 513–520.

Kurt Hornik. 1991. Approximation capabilities of multilayer feedforward networks. *Neural Networks* 4, 2 (1991), 251–257.

J. M. E. Hyland. 2017. Classical lambda calculus in modern dress. *Math. Structures Comput. Sci.* 27, 5 (2017), 762–781.

Teijiro Isokawa, Tomoaki Kusakabe, Nobuyuki Matsui, and Ferdinand Peper. 2003. Quaternion Neural Network and Its Application. In *Proceedings of KES, Part II*. 318–324.

Yann LeCun. 2018. Deep Learning est mort. Vive Differentiable Programming! (2018).

Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation* 1, 4 (1989), 541–551.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

Barak A. Pearlmutter. 1995. Gradient calculations for dynamic recurrent neural networks: a survey. *IEEE Trans. Neural Networks* 6, 5 (1995), 1212–1228.

Barak A. Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a Functional Framework: Lambda the Ultimate Backpropagator. *ACM Trans. Program. Lang. Syst.* 30, 2, Article 7 (March 2008), 36 pages.

J.K. Pearson and David L. Bisset. 1992. Back Propagation in a Clifford Algebra. In *Proceedings of ICANN, Vol. 2*. 413–416.

Gordon Plotkin. 2018. Some Principles of Differential Programming Languages. (2018). https://popl18.sigplan.org/details/POPL-2018-papers/76/Some-Principles-of-Differential-Programming-Languages Invited talk at POPL 2018.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI Blog* 1, 8 (2019).

David E. Rumelhart, James L. McClelland, and PDP Research Group. 1987. *Parallel Distributed Processing, Volumes 1 and 2*. MIT Press.

Aliaksei Severyn and Alessandro Moschitti. 2015. Twitter sentiment analysis with deep convolutional neural networks. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 959–962.

Ronald Van Iwaarden. 1993. Automatic Differentiation Applied to Unconstrained Nonlinear Optimization with Result Verification. *Interval Computations* 3 (1993), 41–60.

Fei Wang, Daniel Zheng, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. 2019. Demystifying differentiable programming: shift/reset the penultimate backpropagator. *PACMPL* 3, ICFP (2019), 96:1–96:31.