∂ is for Dialectica: typing differentiable programming

Marie Kerjean & Pierre-Marie Pédrot
Inria & LS2N

At the core of automatic differentiation (AD) lies the choice of an evaluation strategy for differentials, and in particular for the differential of a composition of functions. In machine learning, reverse-mode AD is mostly implemented in imperative languages, e.g. C++/Python for the TensorFlow and PyTorch libraries. The recent industrial success of these methods has triggered a new research area from the community of researchers in programming language theory. Differentiable programming explores the syntax and the semantics of programming languages endowed with differential transformations. This provides proofs of soundness [1], complexity results [3] or the encoding of AD through primitive programming functions [11, 6].

We take in this paper the point of view of *type theory*. Until now, differentiable programming has mostly been typed within minimal logic with pairs. We instead use linear logic, specifically its differential variant DiLL, to get a linearly-typed view on differentiable programming. This abstract has two aims. The first is to highlight the fact that *the Dialectica transformation is a differential transformation*. We will insist that it has a *reverse-mode* flavour. The second object of this abstract is work in progress, showing that DiLL types a higher-order principled differentiable programming language in which different automated reduction strategies could be expressed.

1 Higher-Order Automatic Differentiation

Automatic Differentiation basically consists in the differentiation of *nested deterministic algebraic expressions*, with an emphasis on efficiency. One has to choose an order of execution for the intermediate computations in an expression, which typically matters for the *chain rule*, i.e. the differentiation of compositions of functions

$$D_x(g \circ f) = D_{f(x)}(g) \circ D_x(f).$$

Assuming x, f and g are given as input, the computation of the linear function $D_x(g \circ f)$ thus requires the intermediate computation of $f(x)$, $D_{f(x)}(g)$ and $D_x(f)$. The order in which those subcomputations are performed is critical.

- *Forward-mode* consists in systematically computing $D_x(f)$ before $D_{f(x)}(g)$: the computation of $f(x)$ and $D_x(f)$ is done in a single forward pass on the algebraic expression.
- *Reverse-mode* consists in systematically computing $D_x(f)$ after $D_{f(x)}(g)$: the computation of $f(x)$ and $g(x)$ is made in a first forward pass on the expression, while the computation of differentials $D_{f(x)}(g)$ and $D_x(f)$ is done in a reverse phase. With the reverse phase, differentiation is contravariant.

This has recently properly been encapsulated by Brunel, Mazza and Pagani in the linear substitution calculus [3]. The differential argument is typed by a linear negation. We recall below the core of their reverse differential transformation on a function $f : \mathbb{R}^n \to \mathbb{R}^m$:

$$\overleftarrow{D}(f) : (a,x) \in \mathbb{R}^n \times \mathbb{R}^{m\perp} \mapsto (f(a), (v \mapsto v \cdot (\mathrm{D}_a f \cdot x)) \in \mathbb{R}^m \times \mathbb{R}^{n\perp}$$

Submitted to:

where $\cdot$ stands for the scalar product in $\mathbb{R}^m$ and $\mathbb{R}^{m\perp}$ stands for the dual of $\mathbb{R}^m$: $\mathbb{R}^{m\perp} := \mathbb{R}^m \multimap \mathbb{R}$.

Higher-order At higher-order, that is for functions between arbitrary topological vector spaces $f : E \to F$, the previous backpropagation formula generalizes to the following expression:

$$\overleftarrow{D}(f) : (a, \ell) \in E \times F' \mapsto (f(a), (\ell \mapsto (\ell \circ \mathrm{D}_a f)) \in F \times E'$$

This makes the differentiation of composed functions at higher-order [3] impossible. Both the differential λ-calculus and Dialectica make differentiation at higher-order possible thanks to *two separate differential transformations*. In the differential λ-calculus [4], a differential expression $Dt \cdot a$ is added to the syntax of λ-calculus, which is computed through an algebraic linear subsitution $\frac{\partial t}{\partial x} \cdot a$. Meanwhile, Dialectica relies on two mutually recursive syntactic translations. We will argue that while differential λ-calculus does this with a forward-mode flavour, Dialectica does it with a reverse-mode flavour.

Ressources In addition to the determinization of the chain rule, AD focusses the optimization of resources. As such, one must avoid to compute a value more than once. Most differentiable programming language thus distinguish differentiation on *primitive function variables* and differentiation on higher-order functions [3, 1, 11]. Differentiation on function variables obeys the reverse chain rule exposed above, while differentiation on higher-order terms obeys a functorial differentiation law:

$$\overleftarrow{D}(tu) = \overleftarrow{D}(t)\overleftarrow{D}(u).$$

This surprising distinction is what enables an efficient computation of nested algebraic expressions. We offer here a point of view from linear logic: higher-order computations are linear in the the sense of linear logic, and differentiation is functorial on linear maps. Consider $\ell : F \multimap G$ linear and $f : E \to F$. Then for any $y \in F$ one has of course $\mathrm{D}_y \ell = \ell$, and as such

$$D_x(\ell \circ f) = \ell \circ D_x f = D_x \ell \circ D_x f.$$

As such, linear logic one can offer a unified perspective with a functorial differentiation, and a language distinguishing between linear and non-linear maps. Taking our inspiration from the smooth semantics of differential linear logic [8], we express this point of view in section 2: as in DiLL, differentiation is functorial and the proper chain rule is expressed by differentiating the promotion rule.

2 ∂ is for Dialectica

Dialectica was originally introduced by Gödel [7] as a way to constructively interpret an extension of HA^ω [2]. It has a strong connection with Linear Logic (LL) [9], and in its intuitionistic version, it consists in two inductively defined transformation on terms of the λ-calculus. We argue that one corresponds to a *partial substitution* on terms while the other one is a *reverse automated differential transformation*. These two transformations are a kind of a reverse-mode account of differential λ-calculus [4]. While the differential λ-calculus is typed by minimal logic, Dialectica [10] is a transformation between two *different* logical systems. The type system at the target is indeed enriched with *an abstract multiset type-former* $\mathfrak{M}$, i.e. a monad equipped with a monoid structure compatible with the monadic operations. In our case, we will use the free vector space monad, which allows to naturally handle differentiation on positive types.

$$
\begin{array}{lcl}
\mathbb{W}(A \Rightarrow B) & := & \begin{pmatrix} (\mathbb{W}(A) \Rightarrow \mathbb{W}(B)) \\ \times \\ (\mathbb{W}(A) \Rightarrow \mathbb{C}(B) \Rightarrow \mathfrak{M}\mathbb{C}(A)) \end{pmatrix} \\
\mathbb{C}(A \Rightarrow B) & := & \mathbb{W}(A) \times \mathbb{C}(B) \\
\mathbb{W}(A \times B) & := & \mathbb{W}(A) \times \mathbb{W}(B) \\
\mathbb{C}(A \times B) & := & \mathbb{C}(A) + \mathbb{C}(B) \\
\mathbb{W}(\mathbb{R}) & := & \mathbb{R} \\
\mathbb{C}(\mathbb{R}) & := & 1
\end{array}
\qquad
\begin{array}{lcl}
x^\bullet & := & x \\
x_x & := & \lambda \pi. \{\pi\} \\
x_y & := & \lambda \pi. \varnothing \qquad \text{if } x \neq y \\
(\lambda x. t)^\bullet & := & (\lambda x. t^\bullet, \lambda x \pi. t_x\ \pi) \\
(\lambda x. t)_y & := & \lambda \pi. (\lambda x. t_y)\ \pi.1\ \pi.2 \\
(t\ u)^\bullet & := & (t^\bullet.1)\ u^\bullet \\
(t\ u)_y & := & \lambda \pi. (t_y\, (u^\bullet, \pi)) \circledast ((t^\bullet.2)\, u^\bullet\, \pi \ggg u_y)
\end{array}
$$

Figure 1: The Revised Dialectica Translation (excerpt)

Proposition 1. *If $\Gamma \vdash t : A$, then $\mathbb{W}(\Gamma) \vdash t^\bullet : \mathbb{W}(A)$ and for any $x : X \in \Gamma$, $\mathbb{W}(\Gamma) \vdash t_x : \mathbb{C}(A) \to \mathfrak{M}\mathbb{C}(X)$.*

Proposition 2. *If $t \equiv_\beta u$, then $t^\bullet \equiv_\beta u^\bullet$ and $t_x \equiv_\beta u_x$ for any x, assuming straightforward equational rules for multisets.*

Proposition 3. *Assuming t is a source function, let us evocatively and write $t' := t^\bullet.2$. Let f and g be two terms from the source language and x a fresh variable. Then, writing $f \circ g := \lambda x. f\ (g\ x)$, we have in the target*

$$(f \circ g)'\ x \equiv \lambda \pi. ((f'\ (g\ x)^\bullet\ \pi) \ggg (g'\ x)).$$

Abstract multisets are to semirings what monads are to monoids. In particular, the $\ggg$ operation is a generalization of the semiring multiplication, so that the above equation can be understood as a generalization of the chain rule. When $f : \mathbb{R}^n \to \mathbb{R}^m$ and $g : \mathbb{R}^p \to \mathbb{R}^n$ it is actually isomorphic to it, because when $\mathfrak{M}(-)$ is the free $\mathbb{R}$-vector space monad, $\ggg$ amounts to matrix multiplication on base types.

Assume a set of primitive differentiable functions $\{f_i : \mathbb{R}^{m_i} \to \mathbb{R}^{n_i}\}_{i \in I}$ in the source language, it is possible to extend the translation trivially as $f_i^\bullet := (f_i, (f_i)')$, abusing the isomorphism $\mathbb{C}(\mathbb{R}^m) \to \mathfrak{M}(\mathbb{C}(\mathbb{R}^n)) \cong \mathbb{R}^{nm}$. As such, we have the following:

Theorem 1. *Dialectica implements* AD *for a higher-order language.*

As Dialectica preserves full β-equivalence, the fact that it operates reverse-mode can only be observed through the types of the differentiated terms. By reintroducing the linear negation into the type system, one gets a contravariant differential transformation. Indeed, when $\Gamma \vdash t : !A \multimap B$ and writing $Dt = (t^\bullet.2)$ we have:

$$\Gamma \vdash Dt : A \Rightarrow (B^\perp \Rightarrow \mathfrak{M}(A^\perp)) \qquad\qquad \Gamma \vdash t_y : A \times B^\perp \Rightarrow \mathfrak{M}(Y^\perp)$$

The differential features in Dialectica tighten the relation between differentiation and delimited continuations, as observed already [11]. It would seem that differentiation allows for the validation of semi-classical axioms such as Markov's Principle. It also shows the relevance of linear logic to the study of DiLL, which we explore more specifically now in the setting of DiLL.

One could forget about the $\mathfrak{M}$ construction and take the differentiation λ-calculus as target language, where:

$$[\![t_x]\!] := \lambda \pi. \frac{\partial [\![t]\!]}{\partial x} \cdot \pi$$

Then one would have $Dt \cdot v = (t^{\bullet}.2)v$. However, the nice proof-theoretical properties of Dialectica are ensured by an addition that is not performed on every type. We reinterpret this in the co-structural rules of DiLL.

3 A term language for DiLL: AD as reduction strategies

Differentiation has in fact been studied as a primitive rule of linear logic by Ehrhard and Regnier [5]. Differential Linear Logic (DiLL) endows ! with co-structural laws alike the one of $\mathfrak{M}$. DiLL adds to it an internal differentiation operator which combine a linear argument with a non-linear one and acts.

In smooth models of (classical) DiLL, the exponential is interpreted as a space of distribution with compact support:

$$[\![!A]\!] := \mathscr{C}^{\infty}([\![A]\!], \mathbb{R})'.$$

This offers an alternative intuition than the quantatitive one on ressources. The exponential in LL allows to build linear transformations into internal object of the language. In DiLL, it allows to internalize differentiation by introducing it through a new exponential rule. We develop a polarized term-language Λ_{AD} for Differential Linear Logic by making explicit the use of distributions and the algebraic operations they can be endowed with. As suggested by Dialectica, sums only happen on some specific types. DiLL highlights that these are done on positive exponentials ! via co-dereliction, while pointwise multiplication is done on negative exponentials ? via structural rules.

$$\begin{gathered} u, v := x \mid t^{\perp} \mid u * v \mid \emptyset \mid u \otimes v \mid 1 \mid \delta_u \mid D_u(t) \mid \downarrow t \\ t, s := u^{\perp} \mid t \cdot s \mid w_1 : N \mid \lambda x.t \mid dx.t \mid \uparrow u \end{gathered}$$

Building on the smooth semantics of DiLL, non-linear arguments are encoded through diracs:

$$\delta_u := f \mapsto f(u) \qquad\qquad (\lambda x.t)(\delta_u) \to t[u/x]$$

While usual arguments are introduced through a promotion rule, linear arguments are introduced through co-derelections.

$$\frac{\vdash \mathscr{N}, t : \uparrow P \mid \qquad \vdash \mathscr{M} \mid u : !P}{\vdash \mathscr{N}, \mathscr{M} \mid D_u t : !P}\ \mathbf{D} \qquad\qquad \frac{\vdash ?\mathscr{N} \mid t : P}{\vdash ?\mathscr{N} \mid \delta_t : !P}\ \mathbf{p}$$

The denotational semantics of $D_u v$ is the distribution mapping a function f to its differential at u, according to the vector v. In the spirit of DiLL, this means that terms are not differentiated as such, but differentiation is introduced as a distribution, that is a mixed linear/non-linear argument typed by !.

$$D_u v := f \mapsto \mathrm{D}_u t(f) \qquad\qquad (\lambda x.t) D_u v \to \text{defined by induction on t}$$

The abstract multiset operation on Dialectica is encoded through *a double exponential*. This shifts Dialectica's sums, which compute in the codomain $\mathfrak{M}(A)$ of a function to a notion of sums defined in the domain of a function. Pédrot's abstract multiset operations can be soundly interpreted in the co-structural rules of Differential Linear Logic: addition is typed by a co-contraction rule, interpreted as the convolution between distributions. The term $(dx.[f]x) : ?N$

denotes the function $f : N$ which has undergone a dereliction, meaning whose linearity has been forgotten.

$$[\emptyset] := \emptyset \quad [\{t\}] := (\delta_{\delta_{[t]}}) \quad [u \circledast v] := ([u] * [v]) \quad [m \ggg f] := (dx.[f]x)[m]$$

Indeed, according to their smooth semantics, the usual algebraic operations —that is additions and scalar products— are done on ground elements of type $\mathbb{R}^n$ and propagated at higher-order through contraction and co-contraction.

$$f \cdot g := x \mapsto f(x) \cdot g(x) \qquad u * v := f \mapsto u(y \mapsto v(z \mapsto f(y * z)))$$

$$\frac{\vdash \mathscr{N}, X^\perp \mid u : P \qquad \vdash \mathscr{M}, X^\perp \mid v : P}{\vdash \mathscr{N}, \mathscr{M}, X^\perp \mid u * v : P}\ \bar{c} \qquad \frac{\vdash \mathscr{N}, f : N, g : N \mid}{\vdash \mathscr{N}, f \cdot g : N \mid}\ c$$

While the Dialectica translation is purely equational, a syntax for **DiLL** allows to modularly compute differentials according to different reduction strategies. Indeed, the differentiation of a composition $(\lambda y.s) \circ (\lambda x.t)$ at a point $u = \delta_w$ according to a vector r computes as

$$(\lambda y.s)((\lambda x.\delta_t)D_u r)) \to (\lambda y.s)D_{t[u/x]}((\lambda x.t)D_u r)$$

The last equation is the exact translation of the chain rule. Whether we compute first $(\lambda x.t)D_u r$, i.e. the differential of $\lambda x.t$, or proceed immediately with the computation of the differential of $\lambda y.s$ depends of the reduction strategy chosen. One can choose to reduce the linear argument of differentials before differentiating functions (call-by-value), or not (call-by-name).

References

[1] Martin Abadi & Gordon D. Plotkin: *A Simple Differentiable Programming Language. POPL 2020.*

[2] Jeremy Avigad & Solomon Feferman (1998): *Gödel's Functional ('Dialectica') Interpretation.* In: *Handbook of Proof Theory*, Elsevier Science Publishers.

[3] Aloïs Brunel, Damiano Mazza & Michele Pagani (2020): *Backpropagation in the Simply Typed Lambda-calculus with Linear Negation. POPL.* Available at `http://arxiv.org/abs/1909.13768`.

[4] Thomas Ehrhard & Laurent Regnier (2003): *The differential lambda-calculus. Theoret. Comput. Sci.* 309(1-3).

[5] Thomas Ehrhard & Laurent Regnier (2006): *Differential interaction nets. Theoretical Computer Science* 364(2).

[6] Conal Elliott (2018): *The simple essence of automatic differentiation.* In: *Proceedings of the ACM on Programming Languages (ICFP).* Available at `http://conal.net/papers/essence-of-ad/`.

[7] Kurt Gödel (1958): *Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. Dialectica* 12, pp. 280–287.

[8] Marie Kerjean: *A Logical Account for Linear Partial Differential Equations.* In: *LICS 2018.*

[9] Valeria de Paiva (1989): *A Dialectica-like Model of Linear Logic.* In: *Category Theory and Computer Science, Manchester, UK, September 5-8, 1989, Proceedings*, pp. 341–356.

[10] Pierre-Marie Pédrot (2014): *A functional functional interpretation.* In: *CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pp. 77:1–77:10.

[11] Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M. Essertel & Tiark Rompf (2019): *Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator. Proc. ACM Program. Lang.* 3(ICFP), pp. 96:1–96:31.