∂ is for Dialectica: Typing Differentiable Programming

Marie Kerjean, Pierre-Marie Pédrot

∂ is for Dialectica: Typing Differentiable Programming

Marie Kerjean & Pierre-Marie Pédrot

Abstract

Differentiable programming is a recent research area: its objective is to express differentiation as a modular algorithmic transformation on rich programming languages. It is in particular motivated by the various applications of automatic differentiation in machine learning or formal calculus. This work focuses on the typing system used to express differentiation. The first part of this paper is devoted to expressing the Dialectica transformation as a reverse automated differentiation transformation on a higher-order lambda-calculus with positive types. The second part builds on the intuitions provided by Dialectica to provide a lambda-calculus with an internal differentiation operator, with a typing system inspired by Differential Linear Logic, allowing to express backward automatic differentiation as a call-by-name strategy and forward automatic differentiation as a call-by-value strategy. The target language of Dialectica is then given a semantics in smooth models of Differential Linear Logic.

1 Introduction

At the core of automatic differentiation (AD) there is the choice of an evaluation strategy for differentials, and in particular for the differential of a composition of functions. More specifically reverse-mode AD in machine learning is mostly implemented in imperative languages, as TensorFlow in Python [ABC$^+$16] or PyTorch [PGC$^+$17]. However, a common principle amongts theoretical computer scientists is that any programming language should have a functionnal and typed core. This observation has triggered a new research area from the community of researchers in the theory of programming languages. To our understanding, differentiable programming explores the syntax and the semantics of programming languages endowed with differential transformations. This allows proofs of soundness [AP20], of complexity results [BMP20] or the encoding of automatic differentation algorithms through more primitive programming functions [WZD$^+$19, Ell18].

We take in this paper the point of view of *type theory*. Until now, differentiable programming were mostly typed with minimal logic with pairs. We use Linear Logic and its Differential refinement to type differentiable programming.

Dialectica was originally introduced by Gödel [G$\ddot{\text{o}}$58] as a way to constructively interpret an extension of HA^ω [AF98]. It has a strong connection with Linear Logic (LL) [dP89], 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-AD account of differential λ-calculus [ER03]. As in the differential λ-calculus, thaving two separate differential transformations allows for the differentiation of higher-order terms. The main exception between that our version of Dialectica [Péd14] is a transformation between two *different* logical system. The type system at the target is indeed enriched with *an abstract multiset operation* $\mathfrak{M}$. This allows to handle differentiation on positive and dependant types.

Differentiation has in fact been studied as a primitive rule of Linear Logic by Ehrhard and Regnier [ER06]. Differential Linear Logic (DiLL) endows !, the traditional exponential object of LL which encodes the non-linearity of proofs, 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. Its denotational interpretation is a linear form which acts on functions, that is a distribution :

$$D_u t : (f : E \Rightarrow F) \mapsto ((D_u f)t : \mathbb{R})$$

where $D_u f$ denotes the differential of f at the point t. We develop a term-language Λ_{AD} for Differential Linear Logic by making explicit the use of distributions and the algebraic operations they use. Building on the semantics of Differential Linear Logic, arguments are encoded through diracs:

$$\delta_u : f \mapsto f(u).$$

Finally, we give a sound denotational semantic to the target language of Pédrot's Dialectica in Λ_{AD} and in a smooth model of Differential Linear Logic.

To our knowledge, it is the first time that the differential features of the Dialectica transformation are identified, making it the first differential transformation to operate on positive and dependant types. Moreover, Λ_{AD} seems to be the first λ-calculus expressing both forward and reverse automatic differentiation it its reduction rules.

Outline We begin the paper by giving an introduction to automated differentiation (Section 2.1) and differentiable programming (Section 2.2). Section 3 is devoted to the study of Dialectica from a differential perspective. Section4 constructs a term language for DiLL, after introducing DiLL and its semantics in Section 4.1. Section 5 finally expresses the target language of Dialectica in Λ_{AD}.

Related work This work is in the line recent work on differentiable programming languages, which are detailed in Section 2.2. It is also close from term languages typed by (polarized) LL [BBPH93, Mun09, KPB15, ET19]. The main difference is that we handle syntactic constructions corresponding exactly to structural rules of LL, and make use of the co-structural exponential rules of DiLL. As such, the reduction rules of Λ_{AD} for the convolution are similar to the one of the convolution $\bar{\lambda}\mu$-calculus [Vau07], with the exception that in the latter they operate in an untyped setting.

Notation 1. Borrowing notations from LL, we write $E \multimap F$ the space of (continuous) linear maps between two (topological) vector spaces. We denote by $- \cdot -$ the pointwise multiplication of reals or real functions.

2 Differentiation in programming languages

2.1 Automatic differentiation

We give here an introduction oriented towards differential calculus and higher-order functional programming. Thus our presentation is at first free from partial derivatives and Jacobians notations. We refer to [BPRS17] for fuller introduction to automatic differentiation. Let us recall the chain rule: consider two differentiable functions $f : E \longrightarrow F$ and $g : F \longrightarrow G$.

$$D_t(g \circ f) = D_{f(t)}(g) \circ D_t(f).$$

When computing the value of $D_t(g \circ f)$ at a point $v : \mathbb{R}^n$ one must determine in which order the following computations must be performed: $f(t)$, $D_t(f)(v)$, the function $D_{f(t)}(g)$ and finally $D_{f(t)}(g)(D_t(f)(v))$. The first two computations are independent from the other ones.

In a nutshell, reverse-mode automatic differentiation[1] consists in computing first $f(t)$, then $g(f(t))$, then the function $D_{f(t)}(g)$, then computing $D_t(f)$ and lastly computing the application of $D_{f(t)}(g)$ to $D_t(f)$. Forward differentiation consists in computing first $f(t)$, then $D_t(f)$, then $g(f(t))$, then $D_{f(t)}(g)$ and lastly applying of $D_{f(t)}(g)$ to $D_t(f)$. This explanation is the one which fits our higher-order functional setting: for a diagrammatic interpretation, see for example [BMP20].

These two techniques have different efficiency when one considers the dimension of E and F as vector spaces. For more specific case of differentiable functions between Euclidean spaces -which is the one on which techniques of automatic differentiation are implemented - one can describe these two algorithms with partial derivatives. Consider $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ and $g : \mathbb{R}^m \longrightarrow \mathbb{R}^k$.

Let us denote $x_1, \ldots, x_n$, $y_1, \ldots, y_m$ and $w_1, \ldots, w_k$ the canonical basis of $\mathbb{R}^n$, $\mathbb{R}^m$ and $\mathbb{R}^k$ respectively. We write f_j (resp. g_l) for the j-th projection of f. We write $(t_1, \ldots, t_i, \ldots, t_n)$ for the projections of t on the canonical basis of $\mathbb{R}^n$. Thus $f_j : \mathbb{R}^n \longrightarrow \mathbb{R}$ and $g_l : \mathbb{R}^m \longrightarrow \mathbb{R}$. The chain rules rewrites as follows:

[1] Backpropagation is an adaptation of reverse-mode to the very specific setting of neural networks

$$\frac{\partial(g \circ f)}{\partial x}(t) = \sum_j \frac{\partial g}{\partial y_j} \cdot (f(t)) \frac{\partial f_j}{\partial x} \cdot t$$
$$= \sum_j \sum_i \frac{\partial g}{\partial y_j} \cdot (f(t)) \cdot \frac{\partial f_j}{\partial x_i} \cdot t_i \cdot (\partial x_i.x)$$

Automatic Differentiation (AD) is in fact differentiation of *nested deterministic algebraic expressions*. One is going to choose an order of execution for the intermediate computations in $g \circ f$, and more precisely for the way the outputs f_j of f are going to be used by g. What makes the difference is that a computation in forward-mode automatic differentiation pass is typically going to be initialized with respect to one canonical real input variable x_i. Each application of the forward-mode automatic differentiation algorithm will compute the derivative of $\frac{\partial g_l}{\partial x_i} \cdot t$ for all output g_l but a single input x_i. Thus the forward-mode automatic differentiation algorithm is computing derivative from inside to outside. It is mainly efficient for programs with few inputs and many outputs: $k \gg n$.

On the other hand, the reverse-mode AD algorithm computes derivatives from outside to inside, and is more efficient for programs with few outputs and many inputs: $n \gg k$. It is going to be initialized with respect to one output derivative y_j. After a first pass computing the values of $f_j(t)$ and $g_l(f(t)$, the reverse derivative pass will compute derivatives of partial computations $\frac{\partial g}{\partial y_j} \cdot (f(t))$ and $\frac{\partial f_j}{\partial x} \cdot t$ for finally computing their multiplication. Let us insists that the use of pointwise multiplications and projections here is very specific to the Euclidean field.

Thus reverse mode AD is particularly well-fitted for several methods in machine learning where one tries to optimize several parameters x_i with respect to the variation an error y.

2.2 Differential programming languages

We give here a review of a few recent related work on differentiable programming. The differential λ-calculus was introduced by Ehrhard and Regnier [ER03] as a syntactic account for the mathematical theory of differential calculus. To the terms of λ-calculus is added a *differential application* $Ds \cdot u$ which represents the term s linearly applied to u. Linearity is understood through the intuition of call-by-name LL: a linear variable is a variable which is going to be computed exactly one time. It also follows the traditional mathematical intuition, that is *head* variables —acting as functions— are linear: one always have

$$(f + g)(x) = f(x) + g(x)$$

while

$$f(x + y) = f(x) + f(y)$$

asks for a special requirement on f. Differential λ-calculus is a forward higher-order differentibale language whichenriches simply typed λ-calculus with a primitive differential construction $Dt \cdot u$, which represents the differential of a term t fed with u as *linear argument*[2] It obeys the following reduction rule:

$$D(\lambda x.t) \cdot u \to_{\beta_D} \lambda x. \frac{\partial t}{\partial x} \cdot u \tag{1}$$

The term $\frac{\partial t}{\partial x} \cdot u$ is the linear substitution of x by u in t and has an operational semantics similar to the partial differentiation operation. It is performed as a static transformation on terms which distributes over sums, recalled in Section 3.4.

Abadi and Plotkin [AP20] recently gave a syntax for a language with a first-order reverse differentation operator. The operational semantics of this operator is based on the computation of the *trace* of a program. They show adequacy with respect to a real-analysis semantics. Eliott [Ell18] gave in a series of paper an account for automatic differentiation in Haskell as a cartesian functor. Wang and al. [WZD+19] give a language with a compositional symbolic reverse-mode AD operator, based on continuation passing style. In this way give a rich operational semantics, although it is symbolic and uses dynamic binding.

[2]The notations of differential λ-calculus are not the one of usual analysis. In analysis one fixes the (non-linear) point at which a function must be differentiated, and the linear argument is latter on fed to the differential of the function. In differential λ-calculus, and latter on in this paper, linearity of the argument means it must substitute a single (linear) occurrence of the variable. Thus, the linear argument of the differential must come before its non-linear argument.

Most importantly for us, Brunel Mazza and Pagani recently refined the work by Wang and al. using a linear negation on ground types, and prove complexity results. What we present here is quit close. Let us note that however their work relies mainly on computational graphs while ours is directed towards type system and functional analysis. At the core, their differential transformation acts on pairs (as in most of the litterature [AP20] [WZD+19] [Ell18]) in the linear substitution calculus [Acc18], so as to make it compositional. Consider $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ differentiable. Then for every $a \in \mathbb{R}^n$, one has a linear map $D_a f : \mathbb{R}^n \multimap \mathbb{R}^m$, and the forward differential transformation has the following type

$\overrightarrow{D}(f) : (a, x) \in \mathbb{R}^n \times \mathbb{R}^m \mapsto (f(a), D_a f \cdot x) \in \mathbb{R}^n \times \mathbb{R}^m$ where $- \cdot -$ represents the scalar product.

In backward mode, their transformation also acts on pairs, but with a contravariant second component, encoded via a linear dual $(-)^\perp$. The notation $(-)^\perp$ is borrowed from LL, where the (hence linear) negation is interpreted denotationaly as the dual on $\mathbb{R}$-vector spaces:

$$[\![A^\perp]\!] := \mathcal{L}([\![A]\!], \mathbb{R}).$$

Thus, an element of $A^\perp$ is a map which computes linearly on A to return a scalar in $\mathbb{R}$.

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

This encodes backward differentiation as, during the differentiation of a composition $g \circ f$, the contravariant aspect of the second component will make the derivative of g be computed before the derivative of f. However, the fact that the first member is covariant while the second is contravariant makes it impossible to lift this transformation to higher-order. Indeed, when one considers more abstractly function between (topological) vector spaces: $f : E \longrightarrow F$, one has:

$$\begin{aligned} \overleftarrow{D}(f) : E \times F' &\to F \times E' \\ (a, \ell) &\mapsto (f(a), (v \in \mapsto (v \cdot (\mathrm{D}_a f \cdot x))) \end{aligned}$$

Consider $g : F \longrightarrow G$. Then $\overleftarrow{D}(f) F \times G' \longrightarrow G \times F'$. If G and F are not self-dual, there is no way to define the composition of $\overleftarrow{D}(f)$ with $\overleftarrow{D}(g)$. Thus higher-order differentiation must be attained with *two different differential transformations*. This is the case in the differential λ-calculus for the forward AD or the Dialectica Transformation for reverse AD, as we show in Section 3.

Synthesis We would like to draw a few general conclusion from the state of the art. Differentiable programming languages are either:

- Either first-order languages with a primitive differential operator but which operates only on function variables [AP20].
- Or higher-order languages with a differentiation transformation, which applies only to first order terms [BMP20].

They are typed by minimal logic with product and $\mathbb{R}$ as a base type, and linear dual.

3 Dialectica as a term language with two linear transformations

3.1 Dialectica is differentiation

In modern terms, it can be described as a realizability interpretation over an extended λ-calculus able to export intensional content from the underlying terms, i.e. the way variables are used. Through the proof-as-program lens, it is better presented as a program translation from a typed λ-calculus with datatypes into another λ-calculus, together with a realizability relation which will be validated by any well-typed term from the source. In this paper, we will follow the presentation given by Pédrot [Péd14], which relies on a kind of Diller-Nahm variant [Dil74] to preserve the equational theory of the source calculus, and we will not care about the realizability predicate, as we are merely interested in computation, not logic. For the sake of simplicity, we recall the Dialectica translation of the simply-typed λ-calculus below.

Types of the source language are inductively defined as

$$A, B := \alpha \mid A \Rightarrow B$$

and terms are the usual λ-terms endowed with the standard β-reduction.

The target language is a bit more involved, as it needs to feature negative pairs and *abstract multisets*.

Definition 1. An abstract multiset is a parameterized type $\mathfrak{M}\,(-)$ equipped with the following primitives:

$$\frac{}{\Gamma \vdash \varnothing : \mathfrak{M}\,A} \qquad \frac{\Gamma \vdash m_1 : \mathfrak{M}\,A \qquad \Gamma \vdash m_2 : \mathfrak{M}\,A}{\Gamma \vdash m_1 \circledast m_2 : \mathfrak{M}\,A}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \{t\} : \mathfrak{M}\,A} \qquad \frac{\Gamma \vdash m : \mathfrak{M}\,A \qquad \Gamma \vdash f : A \Rightarrow \mathfrak{M}\,B}{\Gamma \vdash m \gg\!= f : \mathfrak{M}\,B}$$

We furthemore expect that abstract multisets satisfy the following equational theory.

Monadic laws

$$\{t\} \gg\!= f \equiv f\ t \qquad t \gg\!= (\lambda x.\, \{x\}) \equiv t$$
$$(t \gg\!= f) \gg\!= g \equiv t \gg\!= (\lambda x.\, f\ x \gg\!= g)$$

Monoidal laws

$$t \circledast u \equiv u \circledast t \qquad \varnothing \circledast t \equiv t \circledast \varnothing \equiv t$$
$$(t \circledast u) \circledast v \equiv t \circledast (u \circledast v)$$

Distributivity laws

$$\varnothing \gg\!= f \equiv \varnothing \qquad\qquad t \gg\!= \lambda x.\, \varnothing \equiv \varnothing$$
$$(t \circledast u) \gg\!= f \equiv (t \gg\!= f) \circledast (u \gg\!= f)$$
$$t \gg\!= \lambda x.\, (f\ x \circledast g\ x) \equiv (t \gg\!= f) \circledast (t \gg\!= g)$$

Formally, this means that $\mathfrak{M}\,A$ is a monad with a semimodule structure over $\mathbb{N}$.

We now turn to the Dialectica interpretation itself, which is defined at Figure 1, and that we comment hereafter. We need to define the translation for types and terms. For types, we have two translations $\mathbb{W}(-)$ and $\mathbb{C}(-)$, which correspond to the types of translated terms and stacks respectively. For terms, we also have two translations $(-)^\bullet$ and $(-)_x$, where x is a λ-calculus variable from the source language.

Theorem 2 (Soundness [Péd14])**.** If $\Gamma \vdash t : A$ in the source then we have in the target

- $\mathbb{W}(\Gamma) \vdash t^\bullet : \mathbb{W}(A)$
- $\mathbb{W}(\Gamma) \vdash t_x : \mathbb{C}(A) \Rightarrow \mathfrak{M}\,\mathbb{C}(X)$ provided $x : X \in \Gamma$.

Furthermore, if $t \equiv u$ then $t^\bullet \equiv u^\bullet$ and $t_x \equiv u_x$.

From [Péd14], it follows that the $(-)_x$ translation allows to observe the uses of x by the underlying term. Namely, if $t : A$ depends on some variable $x : X$, then $t_x : \mathbb{C}(A) \Rightarrow \mathfrak{M}\,\mathbb{C}(X)$ applied to some stack $\pi : \mathbb{C}(A)$ produces the multiset of stacks against which x appears in head position in the Krivine machine when t is evaluated against π.

In particular, every function in the interpretation comes with the intensional contents of its bound variable as the second component of a pair. We claim that this additional data is essentially the same as the one provided in the Pearlmutter-Siskind untyped translation implementing reverse AD [PS08]. As such, it allows to extract derivatives in this very general setting.

Lemma 3 (Generalized chain rule)**.** Assuming t is a source function, let us evocatively and locally 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

$$(f \circ g)'\ x \equiv \lambda \pi.\, (f'\ (g\ x)^\bullet\ \pi) \gg\!= (g'\ x).$$

$$
\begin{array}{lcl}
\mathbb{W}(\alpha) & := & \alpha_{\mathbb{W}} \\
\mathbb{C}(\alpha) & := & \alpha_{\mathbb{C}} \\
\mathbb{W}(A \Rightarrow B) & := & (\mathbb{W}(A) \Rightarrow \mathbb{W}(B)) \times (\mathbb{W}(A) \Rightarrow \mathbb{C}(B) \Rightarrow \mathfrak{M}\,\mathbb{C}(A)) \\
\mathbb{C}(A \Rightarrow B) & := & \mathbb{W}(A) \times \mathbb{C}(B)
\end{array}
$$

$$
\begin{array}{lcl}
x^\bullet & := & x \\
x_x & := & \lambda\pi.\,\{\pi\} \\
x_y & := & \lambda\pi.\,\varnothing \qquad \text{if } x \neq y \\
(\lambda x.\,t)^\bullet & := & (\lambda x.\,t^\bullet, \lambda x \pi.\,t_x\ \pi) \\
(\lambda x.\,t)_y & := & \lambda\pi.\,(\lambda x.\,t_y)\ \pi.\mathtt{1}\ \pi.\mathtt{2} \\
(t\ u)^\bullet & := & (t^\bullet.\mathtt{1})\ u^\bullet \\
(t\ u)_y & := & \lambda\pi.\,(t_y\,(u^\bullet, \pi)) \circledast ((t^\bullet.\mathtt{2})\,u^\bullet\,\pi \gg\!= u_y)
\end{array}
$$

Figure 1: Dialectica Interpretation

$$
\mathbb{W}(\mathbb{R}) := \mathbb{R} \qquad \mathbb{C}(\mathbb{R}) := 1
$$

$$
\varphi^\bullet := (\varphi, \lambda\alpha\,\pi.\,\{() \mapsto \varphi'(\alpha)\}) \qquad \varphi_x := \lambda\pi.\,\varnothing
$$

Figure 2: Dialectica Derivative Extension

It is not hard to recognize this formula as a generalization of the derivative chain rule where the field multiplication has been replaced by the monad multiplication. We do not even need a field structure to express this, as this construction is manipulating free structures, in a categorical sense.

By picking a specific instance of abstract multisets, we can formally show that the Dialectica intepretation computes program differentiation.

Definition 4. We will instantiate $\mathfrak{M}\,(-)$ with the free vector space over $\mathbb{R}$, i.e. inhabitants of $\mathfrak{M}\,A$ are formal finite sums of pairs of terms of type A and values of type $\mathbb{R}$, quotiented by the standard equations. We will write

$$\{t_1 \mapsto \alpha_1, \ldots, t_n \mapsto \alpha_n\}$$

for the formal sum $\Sigma_{0<i\leq n}\ (\alpha_i \cdot t_i)$ where $\alpha_i : \mathbb{R}$ and $t_i : A$.

It is easy to check that this data structure satisfies the expected equations for abstract multisets, and that ordinary multisets inject into this type by restricting to positive integer coefficients.

We now enrich both our source and target λ-calculi with a type of reals $\mathbb{R}$. We assume furthermore that the source contains functions symbols $\varphi, \psi, \ldots : \mathbb{R} \to \mathbb{R}$ whose semantics is given by some derivable function, whose derivative will be written $\varphi', \psi', \ldots$ The Dialectica translation is then extended at Figure 3.1.

The soundness theorem is then adapted trivially.

Theorem 5. The following equation holds in the target.

$$(\varphi_1 \circ \ldots \circ \varphi_n)^\bullet.\mathtt{2}\ \alpha\ ()\ \equiv \{() \mapsto (\varphi_1 \circ \ldots \circ \varphi_n)'(\alpha)\}$$

Proof. Direct consequence of Lemma 3 and the observation that for any two $\alpha, \beta : \mathbb{R}$ we have

$$\{() \mapsto \alpha \times \beta\} \equiv \{() \mapsto \alpha\} \gg\!= \lambda\pi.\,\{() \mapsto \beta\}.$$

□

We insist that the theory is closed by conversion, so in practice any program composed of arbitrary λ-terms that evaluates to a composition of primitive real-valued functions also satisfy this equation. Thus, Dialectica systematically computes derivatives in a higher-order language.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash Dt : B^\perp \Rightarrow (A \Rightarrow \mathrm{Tr}(A^\perp)) \qquad \Gamma \vdash u : A}{\Gamma \vdash ((Dt)u) : B^\perp \Rightarrow \mathrm{Tr}(A^\perp)} \qquad \Gamma \vdash w : B^\perp}{\Gamma \vdash ((Dt)u)w : \mathrm{Tr}(A^\perp)} \qquad \Gamma \vdash u_y : A^\perp \Rightarrow \mathrm{Tr}(Y^\perp)}{\Gamma \vdash u_y \gg\!= (((Dt)u)w) : \mathrm{Tr}(Y^\perp)} \qquad \Xi}{\Gamma \vdash (tu)_y w : \mathrm{Tr}(Y^\perp)} *$$

where Ξ stands for

$$\dfrac{\dfrac{\Gamma \vdash t_y : A \Rightarrow B^\perp \Rightarrow \mathrm{Tr}(Y^\perp) \qquad \Gamma \vdash u : A}{\Gamma \vdash (t_y)u : B^\perp \Rightarrow \mathrm{Tr}(Y^\perp)} \qquad \Gamma \vdash w : B^\perp}{\Gamma \vdash ((t_y)u)w : \mathrm{Tr}(Y^\perp)}$$

Figure 3: Reverse-mode AD in the dialectica translation

3.2 Dialectica is reverse-mode AD

In order to show that the linearized Dialectica transformation corresponds to reverse AD, one needs to understand how the differential applies dynamically. A first solution would have been to look at the Dialectica transformation on Krivine Machine as in [Péd14]. For concision, we choose to weaken the target language by recovering the dual of LL and a type Tr of *traces*, and introduce reduction rules. Construct a target $\lambda^\times$ calculus, with the following typing rules and syntax:

$$A, B := \alpha \mid A \Rightarrow B \mid A \times B \mid A^\perp \mid \mathrm{Tr}(A)$$
$$t, u := x \mid (t)u \mid \lambda x.t \mid (t, u) \mid t \mid u \circledast v \mid \emptyset.$$

$$\dfrac{\Gamma \vdash t : A}{\Gamma \vdash \{t\} : \mathrm{Tr}(A)} \qquad \dfrac{\Gamma \vdash t : \mathrm{Tr}(A) \qquad \Gamma \vdash u : \mathrm{Tr}(A)}{\Gamma \vdash t \circledast u : \mathrm{Tr}(A)}$$

$$\dfrac{}{\Gamma \vdash \emptyset : \mathrm{Tr}(A)} \qquad \dfrac{\Gamma \vdash t : \mathrm{Tr}(A) \qquad \Gamma \vdash u : A \Rightarrow \mathrm{Tr}(B)}{\Gamma \vdash t \gg\!= u : \mathrm{Tr}(B)}$$

Notation 2. We write Dt for $p_2 t^\bullet$ when $t : !A \multimap B$.

Then with the Dialectica transformations one has :

Proposition 6. When $\Gamma \vdash t : !A \multimap B$, we have:

$$\Gamma \vdash Dt : B^\perp \Rightarrow (A \Rightarrow \mathrm{Tr}(A^\perp))$$
$$\Gamma \vdash t_y : A \times B^\perp \Rightarrow \mathrm{Tr}(Y^\perp)$$

Then the differentiation of the composition $t \circ (\lambda y.u)$ of two terms at a point w, encoded as $\lambda y.(tu)_y w$, is typed through the proof-tree in Figure 3.

The term t_y equals $\emptyset$ when y is free in t [Péd14, Proposition 6], which is the case when one is actually computing a composition of functions. If one were to choose a reduction strategy agreeing with $\equiv_\beta$ defined in Section 3.1, the key point consists in the computation of $u_y \gg\!= (((Dt)u)w)$. Thus one must compute the differential of t before computing u_y: the differentials are computed in reverse order and we are facing a reverse differentiation strategy. As Dialectica agrees with a call-by-name abstract machine, this hints that *backward propagation agrees with call-by-name*. We will make this correspondence clear in Section 4 by interpreting $\mathrm{Tr}(A)$ as $!!A$.

3.3 Higher dimensions

It is well-known that Dialectica also interprets negative pairs, whose translation will be recalled here. Quite amazingly, they allow to straigthforwardly provide differentials for arbitrary functions $\mathbb{R}^n \to \mathbb{R}^m$.

$$\begin{array}{lcl}
\mathbb{W}(A+B) & := & \mathbb{W}(A)+\mathbb{W}(B) \\
\mathbb{C}(A+B) & := & (\mathbb{W}(A)\to\mathfrak{M}\,\mathbb{C}(A))\times(\mathbb{W}(B)\to\mathfrak{M}\,\mathbb{C}(B)) \\
\mathbb{W}(\forall\alpha.\,A) & := & \forall\alpha_{\mathbb{W}}.\,\forall\alpha_{\mathbb{C}}.\,\mathbb{W}(A) \\
\mathbb{C}(\forall\alpha.\,A) & := & \exists\alpha_{\mathbb{W}}.\,\exists\alpha_{\mathbb{C}}.\,\mathbb{C}(A)
\end{array}$$

Figure 4: Extensions of Dialectica (types only)

Let us write $A \times B$ for the negative product in the source language. It is interpreted directly as

$$\mathbb{W}(A\times B) := \mathbb{W}(A)\times\mathbb{W}(B), \quad \mathbb{C}(A\times B) := \mathbb{C}(A)+\mathbb{C}(B).$$

Pairs and projections are translated in the obvious way, and their equational theory is preserved, assuming a few commutation lemmas in the target [Péd15].

Writing $\mathbb{R}^n := \mathbb{R}\times ...\times\mathbb{R}$ n times, we have the isomorphism

$$\mathbb{C}(\mathbb{R}^n)\to\mathfrak{M}\,\mathbb{C}(\mathbb{R}^m)\cong\mathbb{R}^{nm}.$$

In particular, up to this isomorphism, Theorem 5 can be generalized to arbitrary differentiable functions $\varphi:\mathbb{R}^n\to\mathbb{R}^m$, and the second component of a such function can be understood as an (n,m)-matrix, which is no more than the Jacobian of that function.

Theorem 7. The Dialectica interpretation systematically computes the total derivative in a higher-order language.

3.4 Scaling up

The main strength of our approach lies in the expressivity of the Dialectica interpretation. Due to the modularity of our translation, it can be extended to any construction handled by Dialectica, provided the target language is rich enough. For instance, via the linear decomposition [dP89], the source language can be equipped with inductive types. It can also be adapted to second-order quantification and even dependent types [Péd14]. We sketch the type interpretation for sum types and second-order in Figure 4.

This is in stark contrast with other approaches to the problem, that are limited to weak languages, like the simply-typed λ-calculus. The key ingredient of this expressivity is the generalization of scalars to free vector spaces, as $\mathbb{R}\cong\mathfrak{M}\,1$. The monadic structure of the latter allows to handle arbitrary type generalizations.

Interpreting Dialectica in the differential lambda calculus We recall below the essential rules for Ehrhard and Regnier linear substitution (see Equation 1).

$$\frac{\partial x}{\partial x}\cdot t = t \qquad \frac{\partial y}{\partial x}\cdot t = 0 \qquad \frac{\partial \lambda y.s}{\partial x}\cdot t = \lambda y.\frac{\partial s}{\partial x}\cdot t$$
$$\frac{\partial su}{\partial x}\cdot t = (\frac{\partial s}{\partial x}\cdot t)u + (Ds\cdot(\frac{\partial u}{\partial x}\cdot t))u$$

Dialectica encodes reverse-mode AD directly with the use of linear continuations π. Let us define a direct transformation $[\![-]\!]$ on top of Dialectica, with the differential λ-calculus as target. As Differential calculus is endowed with a minimal type system which does not distinguish a type of *traces*, this makes possible to define the differential transformation as a transformation with Differential λ-calculus as a source and a target.

$$[\![\emptyset]\!] := 0 \qquad [\![t\circledast u]\!] := [\![t]\!]+[\![u]\!] \qquad [\![\{t\}]\!] := [\![t]\!].$$

Proposition 8. Consider two λ-terms t u. Then $[\![t_x]\!]u \equiv \frac{\partial t}{\partial x}\cdot u$ and $((\lambda x.t)^\bullet.2))u \equiv Dt\cdot u$.

4 A lambda-calculus typed by DiLL

While the previous language indeed encodes higher-order reverse-mode AD, it does not handle algebraic expressions at the start nor does it handle an *internal differentiation operator*. We solve this issue by refining the type system to DiLL. This allows in particular to handle forward and backward propagation as reduction choices. This is done at the cost of the modularity described in Section 3.4.

The formulas of DiLL are constructed according to the same grammar as LL, see Figure 5. The negation of a formula A is denoted $A^\perp$ and defined as follows:

$$(!A)^\perp = ?(A^\perp) \quad (A \,\&\, B)^\perp = A^\perp \oplus B^\perp \qquad (A \oplus B)^\perp = A^\perp \,\&\, B^\perp$$
$$(?A)^\perp = !(A^\perp) \quad (A \parr B)^\perp = A^\perp \otimes B^\perp \qquad (A \otimes B)^\perp = A^\perp \parr B^\perp$$
$$1^\perp = \bot \quad \bot^\perp = 1 \qquad 0^\perp = \top \quad \top^\perp = 0$$

We recall the rules for the exponential connectives $\{?, !\}$ of DiLL in Figure 5. The other rules correspond to the usual ones for the MALL group $\{\otimes, \parr, \oplus, \times\}$ [Gir87].

Formulas of DiLL:

$E, F := 0|1|\top|\bot|A^\perp|A \otimes B|A \parr B|A \oplus B|A \times B|!A|?A$

Exponential rules of DiLL:

$$\frac{\vdash \Gamma}{\vdash \Gamma, ?E}\, w \qquad \frac{\vdash \Gamma, ?E, ?E}{\vdash \Gamma, ?E}\, c \qquad \frac{\vdash \Gamma, E}{\vdash \Gamma, ?E}\, d$$

$$\frac{\vdash \Gamma}{\vdash \Gamma, !E}\, \bar{w} \qquad \frac{\vdash \Gamma, !E \qquad \vdash \Delta, !E}{\vdash \Gamma, \Delta, !E}\, \bar{c} \qquad \frac{\vdash \Gamma, E}{\vdash \Gamma, !E}\, \bar{d}$$

$$\frac{\vdash ?\Gamma, E}{\vdash ?\Gamma, !E}\, p$$

Figure 5: Syntax for the formulas and proof-trees of DiLL

Proofs of DiLL are *finite sums* of proof-trees generated by these rules. In particular, the empty sum is a proof. The cut-elimination procedure are detailed for example by Ehrhard [Ehr18] and follow the intuitions for the differentiation in Euclidean spaces.

Semantically, in the category interpreting these proofs as morphisms, it means that the hom-sets must be enriched over commutative monoids, and that any sequent has a zero proof. In the concrete semantics of vector spaces this not a shock, as any vector space has indeed at least an inhabitant, namely its 0 element.

4.1 The language of distributions typed by DiLL

In this section, we give the denotational intuitions on which the language Λ_{AD} (Figures 6 and 7) is constructed.

While traditionally LL is understood in terms of *resources*, DiLL is instead better understood from a *functional analysis point of view*. In a smooth model of DiLL, which enjoys an involutive linear negation, formulas E are interpreted by some topological vector space $[\![E]\!]$. In particular, exponential connectives are interpreted as

$$[\![?E]\!] := \mathcal{C}^\infty([\![E^\perp]\!], \mathbb{R}) \tag{2}$$
$$[\![!E]\!] := \mathcal{C}^\infty([\![E]\!], \mathbb{R})' \tag{3}$$

where $\mathcal{C}^\infty(E, \mathbb{R})$ denotes the space of scalar smooth functions on a topological $\mathbb{R}$-vector space E, and where $E' := \mathcal{L}(E, \mathbb{R})$ denotes a certain topological dual of E. See [Ker18] for a rigorous exposition of these results. The lesson is that *elements typed by* $?E$ *are smooth functions* $f\ g$, while *elements typed by* $!E$ *are distributions* ϕ, ψ, that is linear continuous scalar functions acting on a space of smooth functions.

A crash course on distribution theory We introduce the basic idea of distribution theory without providing any proof nor details on the topology of spaces of functions or distributions. A distribution (with compact support) $\phi \in \mathcal{C}^\infty(E, \mathbb{R})'$ is a linear continuous scalar map acting on smooth functions. The archetypal distribution is the dirac operator δ_x at a point $x \in E$:

$$\delta_x : f \mapsto f(x).$$

One can check easily that δ_x is indeed linear in f. Distributions are sometimes called *generalized functions* as smooth functions with compact support can indeed act as distributions. Consider $g \in \mathcal{C}_c^\infty(\mathbb{R}^n, \mathbb{R})$ such a function, one can consider the distribution with compact support $T_g : f \mapsto \int f(t)g(t)dt$.

No operation generalizing multiplication is available on spaces of distributions [Sch54]. However, *convolution* is an important monoidal symmetric and associative operation than acts on distributions. Given two distributions ϕ and ψ with compact support, one defines the *convolution* $*$ of two distributions as

$$\phi * \psi := f \mapsto \phi(x \mapsto \psi(y \mapsto f(x+y))).$$

This newly defined map is indeed a distribution with compact support. The neutral for the convolution operation is δ_0, the dirac at 0 . As such, one has indeed $\delta_x * \delta y = \delta_{x+y}$. Thus if E is endowed with an commutative associative monoidal operation, so is $!E$. That is, convolution is a higher-order lift of addition[3].

Rules of DiLL **from a dual perspective** These intuitions allow to interpret the rules DiLL adds to LL, and shed a new light on the LL exponential rules. In the perspective of Equations 2 and 3, let us give the interpretation of the exponential rules in terms of functions and distributions. For a formal proof that this is indeed a model of DiLL, see for example previous work by Kerjean [Ker18].

In a categorical model $\mathcal{L}$ of DiLL, the bigebra structure on every object $!E$ is formally derived from the presence a biproduct $\diamond$ on the $\mathcal{L}$ and from the strong monoidality of !, as $!(A \diamond B) \simeq !A \otimes !B$. In distribution theory, this strong monoidality is interpreted as the *Kernel Theorem*. It is deduced by duality from the surjectivity of the morphism (see for example the proof detailed by Treves [Trè67]):

$$f \otimes g \in \mathcal{C}^\infty(E, \mathbb{R}) \otimes \mathcal{C}^\infty(F, \mathbb{R}) \mapsto f \cdot g \in \mathcal{C}^\infty(E \times F, \mathbb{R}).$$

Thus, in a *classical* smooth model of DiLL, the *contraction* c is interpreted by the pointwise multiplication between functions:

$$c_P(f, g) : x \in [\![P^\perp]\!] \mapsto f(x) \cdot g(x) \in \mathbb{R}.$$

The *weakening* w is interpreted the introduction of function constant at 1:

$$w_P := x \in [\![P^\perp]\!] \mapsto 1_{\mathbb{R}}.$$

The *co-contraction* $\bar{c}$ is interpreted by the convolution between two distributions and the *co-weakening* by the dirac at 0.

$$\bar{c}_N(\phi, \psi) := \phi * \psi \in !N$$
$$\bar{w}_N := \delta_0 \in !N.$$

The dereliction maps a linear function to itself, seen as a smooth function whose linearity has been forgotten:

$$d_P(\ell) := \ell \in [\![?P]\!].$$

The *promotion is the dirac*: when a proof of the sequent $!\Gamma \vdash A$ is interpreted by a smooth function $f \in \mathcal{C}^\infty([\![\Gamma]\!], [\![A]\!])$, the proof of the sequent $!\Gamma \vdash !A$ deduced from the later by the promotion rule is:

$$p(f) = \delta \circ f := x \in [\![\Gamma]\!] \mapsto \delta_{f(x)}.$$

The *codereliction* $\bar{d}$ allows then, by precomposition on a function from $!E$ to F, that is by a cut rule on a sequent $\vdash ?E^\perp, F$ to find the differential *at* 0 of f, that is a sequent $\vdash E^\perp, F$. Thus the coderelection corresponds to the precomposition by the differentiation operator at 0, denoted D_0:

$$D_0 : v \in E \mapsto (f \mapsto D_0 f v),$$

$$\bar{d}_N(v \in [\![N]\!]) := D_0 v \in [\![!N]\!].$$

Let us insist that these interpretations are valid in the setting of a *classical* model of DiLL, interpreting the *involutive linear negation*.

[3]The authors are grateful to Pr. Panangaden for this remark

4.2 A higher-order differentiable language

In this section we introduce a higher order language with an internal differentiation operator expressing AD in its reduction. Based on the intuitions explained above, the contraction rule of DiLL allows to type addition, and its higher-order version convolution. This allows us to get rid of the ad-hoc sums of proofs of DiLL.

Likewise, the co-contraction rule of LL correspond to multiplication in $\mathbb{R}$ and its higher-order version, the pointwise multiplication of functions.

Polarization Polarization in LL very basically distinguishes between two classes of formulas, the positive and negative ones, motivated by *proof-search* issues that won't be developed here. This distinction can also be observed semantically: Negatives connective operate on spaces of functions while positive connectives preserve spaces of distributions. Thus, in the term language described in Figure 6, *the generalized addition* $*$ *operates on positives* while *the generalized pointwise multiplication* $(\_\cdot\_)$ *operates on negatives*.

We make use of *focused sequents* with a possibly empty stoup distinguishing the positive formula:

$$\vdash t_1 : \mathcal{N}_1, ..., t_n : N_n \mid u : P \qquad \vdash t_1 : \mathcal{N}_1, ..., t_n : N_n \mid u$$

abridged as $\vdash \mathcal{N}, X^\perp \mid P$.

We will use DiLL rules in their *generalized* version (see the sequent calculus LLP [Lau02, 4.1.2]). In a polarized sequent calculus, a rule implying an exponential $!N$ (resp. $?P$) is admissible when considered on any positive formula (resp. negative formula). Semantically, in the language of distributions and functions detailed informally in Section 4.1, this amounts to say that any dual $x^\perp$ is in particular a smooth function, and any variable x is a distribution δ_x acting on linear maps. Based on denotational intuitions, we ask for an additive co-contraction DiLL.

Negation changes the polarity of a formula but also its role: from hypothesis to conclusion and vice-versa in logic, or from context to term and vice-versa in the calculus. We make use of shifts, which are unary connectives changing the polarity of formulas but not its role. They are supposed to be involutive on negatives, as is negation. Note that *involutive linear negation allows to consider all points as functions*, and thus to get rid of constraints on a separate class of function variables as in [AP20].

Typing arguments We describe in Figure 6 the terms and typing rules for Λ_{AD} with internal automatic differentiation. This calculus handles two primary different sorts of exponential objects. On one hand diracs δ_v are typed by a promotion rule and correspond to the usual non-linear substitution rule. On the other hand differential operators $\mathrm{D}_u t$ are typed by the **D** rule, inferred from DiLL by using co-contraction and a co-dereliction. When u is a non-linear argument of type $!N$, t is a linear argument fed to the same function, thus t is typed by N.

An abstraction $\lambda x.t$ can then handle two case of arguments: the first one is a dirac δ_t, which will lead to the usual substitution rule. The other one is $D_u t$, that is a *linear argument* t followed by a non-linear argument u. The second case corresponds, with the intuitions of differential λ-calculus, to the differentiation of $\lambda x.t$ at u fed by the linear term t. The rules for the differentiation must then be local, i.e. are defined inductively on t.

Typing abstractions While the codereliction $\bar{d}$ allows to introduce *linear arguments*, dereliction allows to introduce *linear abstractions*. The d rule of Figure 6 is could be decomposed via a dereliction rule and a $⅋$ of DiLL (see Figure 5). The term $dx.t : P \Rightarrow M$ represents a *linear map* from $[\![P]\!]$ to $[\![M]\!]$. Likewise, the usual abstraction corresponds to the $⅋$ rule of LL. It represents the usual application through the call-by-name translation of Intuitionistic Logic in LL: $A \Rightarrow B := (!A)^\perp ⅋ B$.

A linear classical setting We give a syntactic account of the elimination of double linear negation. In the denotational semantics, it means that any $\phi \in E''$ is in fact the evaluation at point:

$$\mathtt{ev}_x : \ell \in E' \mapsto \ell(x)$$

[4] Thus, for a distribution $\phi : !A$ acting on smooth functions, we can make it an element of $A \simeq A^{\perp\perp}$ by restricting it to linear function $K : A^{\perp}$:

$$d(\phi) := dk^{A^{\perp}}.\phi(dx.kx).$$

Then reduction rule 7 encodes the elimination of double negation.

Neutral types and terms We use $\emptyset$ to denote the neutral for the addition $*$, so as not to clash with the notations of LL, where 0 denotes the neutral for the coproduct. Likewise, we use $\mathbf{1}$ to denote the neutral for the multiplication $- \cdot -$, so as not to clash with the notations of LL, where 1 denotes the neutral for the tensor. In a vectorial model, while $\mathbf{1}$ is a term interpret by the 1 element of a vector space, 1 is a type interpret by field ($\mathbb{R}$ here) underneath the vector space. In models of DiLL, the neutral for $⅋$ denoted as $\perp$ is also interpreted by $\mathbb{R}$.

Reduction rules The reduction rules of Λ_{AD} are introduced in Figure 7. For readability, we sometimes precise the types of the terms involved. They are direct translations of the denotational intuitions behind DiLL. The usual β-reduction rule applies to linear and non-linear abstractions when applied to a dirac δ_u. Differentiation rules are *local* and bear strong similarity with the algebraic subsitution of the differential λ-calculus detailed in Section 2.2. Notice that Equation 18 gives a *symmetrical account of the differentiation for the application*, leaving the actual chain rule to Equation 19. This supports the idea of a decomposition of the application rule in a $\mu\tilde{\mu}$-calculus for DiLL [CH00]. The linearity of the abstraction $dx.t$ is witnessed through Rule 11: when fed with a linear argument s and a non-linear argument u, $dx.t$ takes into account only the linear argument. It is a direct translation of the cut-elimination between a dereliction and a co-dereliction in DiLL. The algebraic rules express nothing but the fact that $*$ acts on arguments of functions while the product $- \cdot -$ acts on functions. Notice Rule 33 expressing the fact that the convolution of differentials is the composition of differentials. The usual substitution is defined inductively on the terms and distributes on operations:

$$\delta_u[v/x] := \delta_{[}v/x] \qquad (D_u s)[v/x] := D_{u[v/x]}s[v/x] \quad (t \cdot s)[v/x] := t[v/x] \cdot s[v/x]$$
$$(u * w)[v/x] := u[v/x] * w[v/x]$$

By induction on the last rule used in a typing derivation, one shows the following.

Theorem 9. Consider t any term (positive or negative) such that $\vdash \Gamma, t : A$. If t' is such that $t \to^* t'$, then $\vdash \Gamma, t' : A$.

Example 10. The usual differentiation of $x \mapsto x^2$ holds:

$$\begin{aligned}(\lambda x.\uparrow x \cdot \uparrow x)(D_u s) \to & \qquad \uparrow(\downarrow(((\lambda x.\uparrow x)D_u s \cdot x[u/x])* \\ & \qquad\quad \downarrow(x[u/x] \cdot (\lambda x.\uparrow x)D_u s)) \\ \to & \qquad \uparrow(\downarrow(s \cdot \uparrow u) * \downarrow(\uparrow u \cdot s)).\end{aligned}$$

Lemma 11. If x is free in t, then for any u and r one has $(\lambda x.t)D_u r \to^* \uparrow\emptyset$.

Example 12. We retrieve the usual chain-rule via the composition of Rules 18 and 19. Consider two terms $\lambda x.t : A \Rightarrow B$ and $\lambda y.s : B \Rightarrow C$. A cut-rule between these two abstractions is not possible as it is, as one must use promotion on the type of t: $(\lambda y.s) \circ (\lambda x.t) := \lambda x.((\lambda y.s)\delta_t)$. Then the differentiation of

[4] Note that `ev` is δ restricted to linear functions.

Terms:

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

Types:

$$P, Q := X \mid 1 \mid !P \mid \downarrow N \mid P \otimes Q$$
$$N, M := X^{\perp} \mid \perp \mid ?N \mid \uparrow P \mid N \parr N$$

Notation:

$$P \Rightarrow M := (!P)^{\perp} \parr M$$

Typing rules:

$$\frac{}{\vdash x^{\perp} : P^{\perp} \mid x : P}\ (\text{ax})$$

$$\frac{\vdash \mathcal{N}, t : P^{\perp} \mid \qquad \vdash \mathcal{M} \mid u : P}{\vdash \mathcal{N}, \mathcal{M}, (t)u : \perp \mid}\ (cut_L)$$

$$\frac{\vdash \mathcal{N}, t : N \Rightarrow M \mid \qquad \vdash \mathcal{M} \mid u : !N}{\vdash \mathcal{N}, \mathcal{M}, (t)u : N \mid}\ (\text{cut})$$

$$\frac{\vdash \mathcal{N}, t : M, x^{\perp} : (!P)^{\perp} \mid}{\vdash \mathcal{N}, \lambda x.t : (!P)^{\perp} \parr M \mid}\ (\lambda)$$

$$\frac{\vdash \mathcal{N}, x^{\perp} : P^{\perp}, t : M \mid}{\vdash \mathcal{N}, dx.t : (!P)^{\perp} \parr M \mid}\ (d)$$

$$\frac{\vdash \mathcal{N} \mid x : P \qquad \vdash \mathcal{M} \mid y : Q}{\vdash \mathcal{N}, \mathcal{M} \mid x \otimes y : P \otimes Q}\ \otimes$$

$$\frac{\vdash \mathcal{N} \mid u : P}{\vdash \mathcal{N}, \uparrow u : \uparrow P \mid}\ \uparrow \qquad \frac{\vdash \mathcal{N}, t : N \mid}{\vdash \mathcal{N} \mid \downarrow t : \downarrow N}\ \downarrow$$

$$\frac{\vdash \mathcal{N} \mid}{\vdash \mathcal{N}, w_1 : N, \mid}\ w \qquad \frac{\vdash \mathcal{N}, f : N, g : N \mid}{\vdash \mathcal{N}, f \cdot g : N \mid}\ c$$

$$\frac{}{\vdash \emptyset : P}\ \bar{w} \qquad \frac{\vdash \mathcal{N}, X^{\perp} \mid u : P \qquad \vdash \mathcal{M}, X^{\perp} \mid v : P}{\vdash \mathcal{N}, \mathcal{M}, X^{\perp} \mid u * v : P}\ \bar{c}$$

$$\frac{\vdash \mathcal{N}, t : N \mid \qquad \vdash \mathcal{M} \mid u : !N}{\vdash \mathcal{N}, \mathcal{M} \mid D_u t : !N}\ \mathbf{D}$$

$$\frac{\vdash ?\mathcal{N} \mid t : P}{\vdash ?\mathcal{N} \mid \delta_t : !P}\ \mathbf{p}$$

Figure 6: A differentiable higher-order calculus Λ_{AD}: terms and types

Non-Linear substitution:

$$(\lambda x.t)\delta_u \to_\beta t[u/x] \quad (4)$$
$$(dx.t)\delta_u \to t[u/x] \quad (5)$$

η-rules:

$$\lambda x.(tx) \to t \quad (6)$$
$$(dx.t)\delta_{d(\phi)} \to \phi(dx.t) \quad (7)$$

Functional Context:

$$\frac{s \to s'}{D_u s \to D_u s'} \quad (8) \qquad \frac{t \to t'}{(t)s \to t'(s)} \quad (9) \qquad \frac{s \to s'}{(t)s \to (t)s'} \quad (10)$$

Differentiation:

$$(dx.t)D_u s \to t[\downarrow s/x] \quad (11)$$
$$(\lambda x.\uparrow x)D_u s \to s \quad (12)$$
$$(\lambda x.dz.t)D_u s \to dz.((\lambda x.t)D_u s) \quad (13)$$
$$(\lambda x.\lambda z.t)D_u s \to \lambda z.((\lambda x.t)D_u s) \quad (14)$$
$$(\lambda x.\uparrow\emptyset_P)D_u s \to \uparrow\emptyset_P \quad (15)$$
$$(\lambda x.w_{1\,N})(D_u s) \to \uparrow\emptyset_{\downarrow N} \quad (16)$$

$$(\lambda x.\uparrow y)D_u s \to \uparrow\emptyset \text{ where } \emptyset : \downarrow N \text{ when } t : N \quad (17)$$
$$(\lambda x.(t)u)D_w s \to \uparrow(\downarrow((\lambda x.t)D_w s)u * \downarrow(t((\lambda x.u)D_w s))) \quad (18)$$
$$(\lambda x.\uparrow\delta_t)D_u s \to (\lambda z.\uparrow(D_z((\lambda x.t)D_u s)))((\lambda x.t)(u))) \quad (19)$$
$$(\lambda x.\uparrow D_w t)D_u s \to (\lambda z.\uparrow D_z((\lambda x.t)(D_u s)))((\lambda x.w)(D_u s)) \quad (20)$$
$$(\lambda x.\uparrow w_1 * w_2)D_u s \to \uparrow(\downarrow(\lambda x.\uparrow w_1)D_u s * \downarrow(\lambda x.\uparrow w_2)D_u s) \quad (21)$$
$$(\lambda x.(t \cdot r))(D_u s) \to \uparrow(\downarrow(\lambda x.t(D_u s)) \cdot (r[u/x]) * \downarrow(t[u/x])) \cdot (\lambda x.r(D_u s)))) \quad (22)$$

Algebraic rules:

$$u * \emptyset \to u \quad (23)$$
$$\emptyset * u \to u \quad (24)$$
$$f \cdot w_1 \to f \quad (25)$$
$$w_1 \cdot f \to f \quad (26)$$
$$(d(x).t)\emptyset \to \emptyset \quad (27)$$
$$(\mathbf{1} : P \Rightarrow \bot)D_u t \to \emptyset \quad (28)$$
$$(\mathbf{1} : P \Rightarrow \bot)\emptyset \to \mathbf{1} \quad (29)$$

$$\delta_u * \delta_v \to \delta_{u*v} \quad (30)$$
$$\delta_u * D_v s \to D_{u*v} s \quad (31)$$
$$D_v s * \delta_u \to D_{v*u} s \quad (32)$$
$$D_u t * D_v s \to D_{D_{u*v}t} s \quad (33)$$
$$(\lambda x.t) \cdot (\lambda y.s) \to \lambda x.(t \cdot s[x/y]) \quad (34)$$
$$(\lambda x.t) \cdot (dx.s) \to \lambda x.(t \cdot s) \quad (35)$$
$$(dx.t) \cdot (dx.s) \to \lambda x.(s \cdot t) \quad (36)$$
$$(dx.t)(u * v) \to \downarrow((d(x).t)u) * \downarrow((d(x).t)v) \quad (37)$$

Algebraic Contexts:

$$\frac{u \to u'}{u * v \to u' * v} \qquad \frac{u \to u'}{v * u \to v * u'} \qquad \frac{f \to f'}{f \cdot g \to f' \cdot g}\,{}^{14} \qquad \frac{f \to f'}{g \cdot f \to g \cdot f'} \qquad \frac{u \to u'}{\uparrow u \to \uparrow u'} \qquad \frac{t \to t'}{\downarrow t \to \downarrow t'}$$

Figure 7: A differentiable higher-order calculus: reduction rules

$(\lambda y.s) \circ (\lambda x.t)$ at a point $u = \delta_w$ according to a vector r computes as follows:

$$\begin{aligned}
(\lambda x.((\lambda y.s)\delta_t))D_u r &\to \uparrow(\downarrow((\lambda x.(\lambda y.s))D_u r)\delta_t * \\
&\quad \downarrow((\lambda y.s)((\lambda x.\delta_t)D_u r))) \\
&\to \uparrow(\downarrow((\lambda y.((\lambda x.s)(D_u r)))\delta_t \\
&\quad * \downarrow((\lambda y.s)((\lambda x.\delta_t)D_u r))) \text{ by Rule 14} \\
&\to^* \uparrow(\downarrow(((\lambda x.s)(D_u r)))[t/y] \\
&\quad * \downarrow((\lambda y.s)((\lambda x.\delta_t)D_u r))) \\
&\to^* \uparrow(\downarrow(\uparrow\emptyset) * \downarrow((\lambda y.s)((\lambda x.\delta_t)D_u r))) \text{ by lemma 11} \\
&\to^* (\lambda y.s)((\lambda x.\delta_t)D_u r)) \text{ by involutivity of the shifts} \\
&\to (\lambda y.s)(\lambda z.\uparrow(D_z((\lambda x.t)D_u r)))((\lambda x.t)(u))) \text{ by rule 19} \\
&\to (\lambda y.s)D_{t[w/x]}((\lambda x.t)D_u r) \text{ as } u = \delta_w
\end{aligned}$$

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). This remark is expanded upon in Section 4.5.

Example 13. We provide examples of typed multiplication and addition through structural rules:

$$\dfrac{\dfrac{\vdash x^\perp : X^\perp \mid x : X}{\vdash x^\perp : X^\perp \mid \delta_x : !X}\,\mathrm{p} \qquad \dfrac{\vdash x^\perp : X^\perp \mid x : X}{\vdash x^\perp : X^\perp \mid \delta_x : !X}\,\mathrm{p}}{\vdash x^\perp : X^\perp; x^\perp : X^\perp \mid \delta_x \otimes \delta_x : !X}\,\bar{c}$$

$$\dfrac{\dfrac{\dfrac{\vdash x^\perp : P^\perp \mid x : P}{\vdash d(x).x : (!P)^\perp}\,d \qquad \dfrac{\vdash x^\perp : P^\perp \mid x : P}{\vdash d(x).x : (!P)^\perp}\,d}{\vdash d(x).x : (!P)^\perp \parr \uparrow P, d(x).x : !(!P)^\perp \parr \uparrow P \mid}\,\mathrm{mix};\parr}{\vdash (d(x).\uparrow x) \cdot (d(x).\uparrow x) : !P^\perp \parr \uparrow P \equiv P \Rightarrow \uparrow P}\,c$$

Normalization We acknowledge that we give no normalization nor confluence proof for this calculus: it should follow from the one of DiLL [Pag09] and will be added to a long version of this paper. Indeed, any reduction rule of the calculus is the direct translation of a cut-elimination rule for DiLL, with the exception that sums of proofs are computed in the language through the generalized $*$. The rules for linear substitution (via the application of differential arguments $D_u v$) are the translation of commutative rules involving the codereliction. We refer to the survey by Erhard [Ehr18] for a detailed exposition of the cut-elimination rules.

Semantics While no shift operation is available, complete metrisable nuclear spaces and their dual provide a first-order model for this calculus. Negative formulas are interpreted as nuclear Fréchet spaces and positive formulas as nuclear DF-spaces[5]. The *only* nuclear spaces with are both Fréchet and DF are the Euclidean spaces $\mathbb{R}^n$. Thus, the Euclidean spaces interpret the types of programs on which one can perform both addition and multiplication, corresponding to the scalar product between vectors of $\mathbb{R}^n$. The heavy use of shifts at higher-order is questionable but necessary to interpret multiplication at higher-order. A solution similar to what is done in Dialectica would be to to encode addition directly in the target through an exponential (see Section 5).

[5]Fréchet spaces are metrisable complete lcs, while DF spaces describe their strong duals. Nuclear spaces are the lcs on which several different topological tensor product correspond. Precise definitions can be found in the literature [Jar81, 12.4, 21.1]

4.3 A differential call-by-push value

4.4 sec:cbpv

4.5 Automatic Differentiation as reduction strategies

In this section we merely formalize the remark issued from example 12. Consider the following term:

$$(\lambda x.t)(D_u(f(D_v s))$$

From Section 2.1 we gather that reducing the above term according to Rules 11 to 22 (depending on t) would amount to a backward differentiation strategy. Reducing instead according to Rule 8 would amount to a forward differentiation strategy.

We thus introduce a class of values defined inductively:

$$V, U := x \mid V^{\perp} \mid \emptyset \mid \mathbf{1} \mid \downarrow u \mid \delta_u \mid \lambda x.t \mid dx.t \mid D_u V$$

We choose to reduce algebraic terms and application left to right, thus restricting the algebraic contexts in the obvious way. We have then two reduction strategies:

Definition 14. On Λ_{AD} defined in Figure 7 call-by-name strategy defined by the non-linear substitution rules, Rules 9 10 and all the differentiation and algebraic rules.

Definition 15. On Λ_{AD} defined in Figure 7 call-by-value strategy defined by the non-linear substitution rules, Rules 8 9 10, the algebraic rules and the differentiation rules *restricted to* $D_u V$ where V is a value as above.

5 Interpreting Dialectica in DiLL and its models

In this section we interpret the target language of the Dialectica transformation into Λ_{AD}, by encoding the multiset operation as a *double exponential*. The same translation gives a denotational smooth semantics for it in the model of convenient vector spaces [BET12].

Let us define a translation on top of Dialectica's (see Figure 1), with target a version of Λ_{AD} enriched with products and without pointwise multiplication:

$$u, v := t^{\perp} \mid u * v \mid \emptyset : P \mid \delta_u \mid D_u(t) \mid \downarrow t \; t, s := x \mid u^{\perp} \mid (t, s) \mid \lambda x.t \mid dx.t \mid \uparrow u \; P, Q := X^{\perp} \mid 1 \mid !P \mid \downarrow N$$
$$N, M := X \mid \perp \mid ?N \mid \uparrow P \mid N \times M$$

with the usual typing rules on pairs:

$$\frac{\vdash \mathcal{N}, t : N \qquad \vdash \mathcal{N}, s : M}{\vdash \mathcal{N}, (t, s) : N \,\&\, M}$$

Then we define an interpretation on types and terms:

$$\mathbb{L}(\alpha_{\mathbb{W}}) := \alpha \qquad \mathbb{L}(\alpha_{\mathbb{C}}) := \uparrow \alpha^{\perp}$$
$$\mathbb{L}(\mathfrak{M}\, A) := \uparrow !! \mathbb{L}(A) \qquad \mathbb{L}(A \times B) := \mathbb{L}(A) \times \mathbb{L}(B)$$
$$\mathbb{L}(A \Rightarrow B) := \downarrow \mathbb{L}(A) \Rightarrow \mathbb{L}(B)$$

$$[x] := x$$
$$[\lambda x.t] := \lambda x.[t] \qquad [(t, u)] := ([t], [u])$$
$$[\emptyset] := \uparrow \emptyset \qquad [\{t\}] := \uparrow (\delta_{\delta_{\lfloor t \rfloor}})$$
$$[u \circledast v] := \uparrow (\downarrow [u] * \downarrow [v]) \qquad [m \gg\!= f] := (dx.[f]x)[m]$$

Proposition 16. If $\Gamma \vdash t : A$ in the target of Dialectica, then :

$$\mathbb{L}(\Gamma) \vdash [t] : \mathbb{L}(A)$$

and if $t \equiv u$ in the target of Dialectica then $[t] \equiv [u]$ in our calculus.

Proof. **Monadic laws**. . Moral: when the image of the promotion (ie diracs) are dense in the exponential, we have a monad.

Monoidal laws Immediate

Distributivity laws The compatibility with the distributivity laws justifies the use of a *double* exponential. Indeed, a function $f : A \to B \equiv !A \mapsto B$ needs to be *linear* with respect to the sum $*$. That is, the sum $*$ needs to operate on object typed by !!. □

This proposition is in fact better understood in a concrete denotational model of DiLL.

We now use the interpretation of Dialectica in DiLL to provide a concrete denotational interpretation of Dialectica in a model of intuitionistic DiLL. We recall below the main characteristic of convenient spaces.

- Formulas are interpreted as real vector spaces with vector bornologies, which moreover verify a certain notion of completion (Mackey completion).
- Linear proofs are interpreted by linear bounded maps. The space of all scalar linear bounded maps on a convenient vector space E is denoted E'.
- Non-linear proofs are interpreted as maps $f : E \longrightarrow F$ smooth for some notion which generalizes the usual notion of smoothness between Euclidean spaces. Spaces of smooth maps from E to F are denoted as $\mathcal{C}^\infty(E, F)$.
- The exponential $!E$ has as basis the set of all dirac $\delta x : \mathcal{C}^\infty(E, \mathbb{R}) \longrightarrow \mathbb{R}$, for all $x \in E$. Thus the interpretation for the structural and co-structural exponential rules of DiLL are defined directly on diracs, and so are the $\gg=$ operations below. They can then be extended by linearity and by the universal property of the completion.

The fundamental isomorphism of LL is verified:

$$\chi \ : \ \mathcal{C}^\infty(E, F) \ \simeq \ \mathcal{L}(!E, F).$$

It is defined such that for all f and t well-typed:

$$\chi(f)(\delta_t) = f(t) \tag{38}$$

We interpret inductively the target language of the dialectica translation on types and terms, exposed in Figure 1, as follows:

$$\begin{aligned} [\![\mathfrak{M}\, A]\!] &:= \mathcal{C}^\infty((\mathcal{C}^\infty([\![A]\!], \mathbb{R})'), \mathbb{R})' \\ [\![A \Rightarrow B]\!] &:= \mathcal{C}^\infty([\![A]\!], [\![B]\!]) \end{aligned}$$

$$\begin{aligned} [\![\emptyset]\!] &:= \delta_{\delta_{0:E}} \qquad [\![(\delta_{\delta_e}) \gg= f]\!] := \chi(f)(\delta_e) \\ [\![\{t\}]\!] &:= \delta_{\delta_{[\![t]\!]}} \qquad [\![u * v]\!] := [\![u]\!] * [\![v]\!] \end{aligned}$$

Thanks to the monoidality of the convolution on sums of diracs, and to Equation 38, one checks inductively:

Proposition 17. Consider two terms t, u in the target language of Dialectica. Then if $t \equiv u$, one has $[\![t]\!] = [\![u]\!]$ in the category of convenient spaces.

6 Conclusion

In this paper we detailed two results on differentiable programming: a first part identifies the linearized Dialectica transformation as a differential transformation, which is the first typed differential transformation acting on dependant and positive types. The second part introduces a differential λ-calculus with backward and forward differentiation, based on dual types for distributions and functions. DiLL acts as a bridge between the two structures, but at this point the Dialectica transformation cannot be made into a endo-transformation on a typed language, nor can the second differentiable language handle richer types.

We believe that in the differences between the two parts lie many exciting research perspectives: how to encode higher-order differentiation in Pédrot's Dialectica abstract multiset transformation ? How to

give a $\mu\tilde{\mu}$-calculus for DiLL and handle co-structural rule explicitely ? Recursive fonctions and probabilities might be added by studying the common points between our language and probabilistic languages in call-by-push-value style [ET19].

This also raises *denotational perspectives*. To our knowledge, no axiomatization exists for categorical models of polarized DiLL, and as such of our calculus. Contraints of higher-order polarization make moreover the construction of a model of Λ_{AD} non-trivial. Chiralities [Mel16] seem to be a strong basis on which to build a categorical axiomatization for polarized model of DiLL, in which the double exponential construction at the target of Dialectica could be implemented

References

[ABC+16] 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. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 265–283, 2016.

[Acc18] Beniamino Accattoli. Proof nets and the linear substitution calculus. In *Theoretical Aspects of Computing - ICTAC 2018 - 15th International Colloquium, Stellenbosch, South Africa, October 16-19, 2018, Proceedings*, pages 37–61, 2018.

[AF98] Jeremy Avigad and Solomon Feferman. Gödel's functional ('dialectica') interpretation. In Samuel R. Buss, editor, *Handbook of Proof Theory*, pages 337–405. Elsevier Science Publishers, Amsterdam, 1998.

[AP20] Martin Abadi and Gordon D. Plotkin. A simple differentiable programming language, 2020.

[BBPH93] Nick Benton, Gavin Bierman, Valeria de Paiva, and Martin Hyland. A term calculus for intuitionistic linear logic. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications (TLCA)*. Springer, January 1993.

[BET12] R. Blute, T. Ehrhard, and C. Tasson. A convenient differential category. *Cah. Topol. Géom. Différ. Catég.*, 2012.

[BMP20] Aloïs Brunel, Damiano Mazza, and Michele Pagani. Backpropagation in the simply typed lambda-calculus with linear negation. *POPL*, 2020.

[BPRS17] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *J. Mach. Learn. Res.*, 18:153:1–153:43, 2017.

[CH00] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. *ICFP '00)*, 2000.

[Dil74] Justus Diller. Eine Variante zur Dialectica-Interpretation der Heyting-Arithmetik endlicher Typen. *Archiv für mathematische Logik und Grundlagenforschung*, 16(1-2):49–66, 1974.

[dP89] Valeria de Paiva. A dialectica-like model of linear logic. In *Category Theory and Computer Science, Manchester, UK, September 5-8, 1989, Proceedings*, pages 341–356, 1989.

[Ehr18] Thomas Ehrhard. An introduction to differential linear logic: proof-nets, models and antiderivatives. *Mathematical Structures in Computer Science*, 28(7):995–1060, 2018.

[Ell18] Conal Elliott. The simple essence of automatic differentiation. In *Proceedings of the ACM on Programming Languages (ICFP)*, 2018.

[ER03] T. Ehrhard and L. Regnier. The differential lambda-calculus. *Theoretical Computer Science*, 309(1-3), 2003.

[ER06] T. Ehrhard and L. Regnier. Differential interaction nets. *Theoretical Computer Science*, 364(2), 2006.

[ET19] Thomas Ehrhard and Christine Tasson. Probabilistic call by push value. *Logical Methods in Computer Science*, 15(1), 2019.

[G$\ddot{\text{o}}$58] Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958.

[Gir87] Jean-Yves Girard. Linear logic. *Theoret. Comput. Sci.*, 50(1), 1987.

[Jar81] Hans Jarchow. *Locally convex spaces*. B. G. Teubner, 1981.

[Ker18] Marie Kerjean. A logical account for linear partial differential equations. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 589–598, 2018.

[KPB15] Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. Integrating linear and dependent types. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 17–30, 2015.

[Lau02] O. Laurent. *Etude de la polarisation en logique.* Thèse de doctorat, Université Aix-Marseille II, March 2002.

[Mel16] Paul-André Melliès. Dialogue categories and chiralities. *Publ. Res. Inst. Math. Sci.*, 52(4):359–412, 2016.

[Mun09] Guillaume Munch-Maccagnoni. Focalisation and classical realisability. In *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*, pages 409–423, 2009.

[Pag09] Michele Pagani. The cut-elimination theorem for differential nets with promotion. In *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, pages 219–233, 2009.

[Péd14] Pierre-Marie Pédrot. A functional functional interpretation. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 77:1–77:10, 2014.

[Péd15] Pierre-Marie Pédrot. *A Materialist Dialectica. (Une Dialectica matérialiste).* PhD thesis, Paris Diderot University, France, 2015.

[PGC+17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[PS08] Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Trans. Program. Lang. Syst.*, 30(2):7:1–7:36, 2008.

[Sch54] Laurent Schwartz. Sur l'impossibilité de la multiplication des distributions. *C. R. Acad. Sci. Paris*, 239:847–8, 1954.

[Trè67] François Trèves. *Topological vector spaces, distributions and kernels.* Academic Press, New York-London, 1967.

[Vau07] Lionel Vaux. Convolution lambda mu-calculus. In *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, pages 381–395, 2007.

[WZD+19] Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.*, 3(ICFP):96:1–96:31, July 2019.