\( \partial \) is for Dialectica

Marie Kerjean  
marie.kerjean@cnrs.fr  
CNRS, LIPN, Université Sorbonne Paris Nord  
France

Pierre-Marie Pédrot  
pierre-marie.pedrot@inria.fr  
Inria  
France

ABSTRACT
Automatic Differentiation is the study of the efficient computation of differentials. While the first automatic differentiation algorithms are concomitant with the birth of computer science, the specific backpropagation algorithm has been brought to a modern light by its application to neural networks. This work unveils a surprising connection between backpropagation and Gödel’s Dialectica interpretation, a logical translation that realizes semi-classical axioms. This unexpected correspondence is exploited through different logical settings. In particular, we show that the computational interpretation of Dialectica translates to the differential \( \lambda \)-calculus and that Differential Linear Logic subsumes the logical interpretation of Dialectica.

KEYWORDS
Dialectica, Automatic Differentiation, Linear Logic, Realizability, Category Theory

ACM Reference Format:

1 INTRODUCTION
Dialectica was originally introduced by Gödel in a famous paper [24] as a way to constructively interpret an extension of higher-order arithmetic [4]. It turned out to be a very fertile object of its own. Judged too complex, it was quickly simplified by Kreisel into the well-known realizability interpretation that now bears his name. Soon after the inception of Linear Logic (LL), Dialectica was shown to factorize through Girard’s embedding of LJ into LL, purveying an expressive technique to build categorical models of LL [16]. Yet another way to look at Dialectica is to consider it as a program translation, or more precisely two mutually defined translations of the \( \lambda \)-calculus exposing intensional information [34]. Meanwhile, in its logical outfit, Dialectica led to numerous applications and was tweaked into an unending array of variations in the proof mining community [29].

In a different scientific universe, Automatic Differentiation [25] (AD) is the field that studies the design and implementation of efficient algorithms computing the differentiation of mathematical expressions and numerical programs. Indeed, due to the chain rule, computing the differential of a sequence of expressions involves a choice, namely when to compute the value of a given expression and when to compute the value of its derivative. Two extremal algorithms coexist. On the one hand, forward differentiation [40] computes functions and their derivatives pairwise in the order they are provided, while on the other hand reverse differentiation [31] computes all functions first and then their derivative in reverse order. Depending on the setting, one can behave more efficiently than the other. Notably, reverse differentiation has been critically used in the fashionable context of deep learning.

Differentiable programming is a rather new and lively research domain aiming at expressing automatic differentiation techniques through the prism of the traditional tools of the programming language theory community. As such, it has been studied through big-steps semantics [1], continuations [39], functoriality [22], and linear types [10, 37]. Surprisingly, these various principled explorations of automatic differentiation are what allows us to draw a link between Dialectica and differentiation in logic.

The simple, albeit fundamental claim of this paper is that, behind its different logical avatars, the Dialectica translation is in fact a reverse differentiation algorithm. In the domain of proof theory, differentiation has been very much studied from the point of view of linear logic. This led to Differential Linear Logic [21] (DiLL), differential categories [8], or the differential \( \lambda \)-calculus. To support our thesis with evidence, we will draw a correspondence between each of these objects and the corresponding Dialectica interpretation.

2 SUMMARY OF THE RESULTS
Our main thesis is that the computational content of the Dialectica interpretation is reverse differentiation. In particular, the Dialectica interpretation of function composition boils down to the chain rule. We would like however to expose immediately the kernel of this correspondence.

2.1 A Dialectica Primer
In its most traditional form, Dialectica acts as some elaborate prenex-ation of formulas from intuitionistic arithmetic \( HA \) into higher-order arithmetic \( HA^\omega \). The soundness theorem of the interpretation can be stated as follows.

\[
\text{If } \vdash_{HA} A \text{ then } \vdash_{HA^\omega} \exists x : W(A), \forall y : C(x). x \perp A y. \quad (1)
\]
This should be read as a realizability model, where \( W(A) \) is the simple type of realizers or witnesses of \( A, C(x) \) the simple type of stacks or counters of \( A \) and \( \perp_A \subseteq W(A) \times C(x) \) the realizability condition for \( A \).

The need for higher-order terms is a staple of realizability, where logical implications are interpreted as some flavour of higher-order functions. By the time it was invented, Dialectica was the first of its kind. Although published quite late, it was indeed designed by Gödel even before the inception of the concept of realizability by Kleene. This primitive essence resulted in Dialectica being
nicknamed the functional interpretation, but retrospectively other realizability models are no less functional. As another side-effect of its antiqueness, Dialectica is somewhat shrouded in a veil of mystery. A good part of this is due to historical craft that can be easily scrambled away. For instance, the original version notoriously relies on sequences of objects for the \( W \) and \( C \) mentioned above. This is just an artifact encoding that can be advantageously replaced by products. For simplicity we will also ditch sequences.

Since we will be presenting a more correct version of Dialectica in Section 4, we will not detail the full historical interpretation here but simply hint at the important facts. While most connectives follow the BHK interpretation in Dialectica, the surprise stems from the interpretation of implication [4].

\[
W(A \Rightarrow B) := \begin{cases} \langle W(A) \Rightarrow W(B) \rangle \\ \langle W(A) \Rightarrow C(B) \Rightarrow C(A) \rangle \end{cases}
\]

\[
C(A \Rightarrow B) := \langle W(A) \times C(B) \rangle
\]

\[
(f, \phi) \downarrow_{A \Rightarrow B} (u, \pi) := u \downarrow_A \phi u \pi \Rightarrow f u \downarrow_B \pi
\]

Contrary to the usual BHK interpretation as implemented in e.g. Kreisel realizability, implications \( A \Rightarrow B \) are not simply interpreted as maps \( W(A) \Rightarrow W(B) \) from witnesses to witnesses, but they additionally carry a function of type \( W(A) \Rightarrow C(B) \Rightarrow C(A) \) mapping counters to counters in the reverse direction.

It can be explained as the least unconstructive way to perform a skeletonization on the implication of two formulas which already went through the Dialectica transformation [29, 8.1]. It is also presented as a way to “extract constructive information through a purely local procedure” [4, 3.3].

The historical presentation of Dialectica is defective, insofar as it is merely a logical interpretation rather than a computational one. By this we mean that it preserves provability, but not in general the equational theory of HA seen as a kind of \( \lambda \)-calculus [35]. In spite of this, it is already good enough to understand our original claim.

2.2 The Chain Rule

The chain rule is the bread and butter of calculus, let alone automatic differentiation. Writing \( D_a f \) for the differentiation of \( f \) at \( a \), the chain rule is the name of the high-school equation

\[
D_a(g \circ f) = D_f(a) \cdot g \circ D_a f
\]

describes differentiation of the composition of two functions in terms of the differentiation of these functions.

Just by looking at its type, one can already forecast that the second parameter in the Dialectica witness of an implication will validate a chain rule. (Equation 3). Let us make this explicit. Let \((f, \phi) : W(A \Rightarrow B) \) and \((g, \psi) : W(B \Rightarrow C) \) two witnesses of the corresponding implication through the Dialectica interpretation. That is, it is the case that for all \( u : W(A), v : W(B), \pi : C(B) \) and \( \rho : C(C) \) we have:

\[
(f, \phi) \downarrow_{A \Rightarrow B} (u, \pi) := u \downarrow_A \phi u \pi \Rightarrow f u \downarrow_B \pi
\]

\[
(g, \psi) \downarrow_{B \Rightarrow C} (v, \rho) := v \downarrow_B \psi v \rho \Rightarrow g v \downarrow_C \rho.
\]

The Dialectica interpretation of the composition provides a solution \((h, \chi) : W(A \Rightarrow C)\) satisfying the system below for any \( u : W(A) \) and \( \rho : C(C) \).

\[
(h, \chi) \downarrow_{A \Rightarrow C} (u, \rho) := u \downarrow_A \chi u \rho \Rightarrow h u \downarrow_C \rho
\]

This solution amounts to the following equations.

\[
h u = g(f u) \quad (2)
\]

\[
\chi u \rho = \phi u (\psi(f u) \rho) \quad (3)
\]

While the first equation represents the traditional functoriality of composition, the second equation is almost exactly the chain rule. Indeed, let us write \( D_a(f, \phi) := \phi a \). Then equation 3 can be written as

\[
D_a(g, \psi) \circ_D (f, \phi) = D_f(a)(g, \psi) \cdot D_a(f, \phi) \quad (4)
\]

where \( \circ_D \) is the Dialectica interpretation of the composition and \( f \circ g := g \circ f \). We discuss this reversal a bit later, but for now, let us assert the main thesis of this paper.

**Thesis 1.** The pair \((f, \phi)\) of Dialectica witness of an implication computes a function \( f \) and its differential \( \phi \).

In analysis, many functional transformations satisfy the chain rule [3]. Our goal is thus to strengthen our claim and leave no doubt to the reader that Dialectica is indeed differentiation.

2.3 Reverse Automatic Differentiation

Let us now focus on the puzzling reversal of the chain rule. Remember that assuming a function \( f : A \rightarrow B \), its differential is usually given the type \( Df : A \rightarrow A \rightarrow B \). The second arrow actually stands for a linear map from \( A \) to \( B \). Indeed, the differential of \( f \) at any point can be defined as the best linear approximation of a function at that point.

By contrast and as we have seen, in Dialectica, the second projection of the witness type of an arrow is slightly different. Indeed, the second arrow in this projection is contravariant, as the second component of \( W(A \Rightarrow B) \) is \( W(A) \Rightarrow C(B) \Rightarrow C(A) \). This is what forces us to write function composition as the chain rule the other way around.

Thankfully, this is a well-understood phenomenon, both from the point of view of differentiation and realizability. If we were to understand \( C(A) \) as a kind of dual of the space \( W(A) \), the second projection of the witness of an arrow would have the type of a reverse differential. This dualization is not surprising from the realizability standpoint either, as \( C(A) \) materializes the type of stacks of \( A \). Stacks appear naturally when in the context of abstract machines, where they are dual to terms. As a matter of fact, stacks play an critical role in classical realizability where they are given a first-class citizenship, and can be typed by types that dualize the types of terms.

More astonishingly, it turns out that this dualization already exists in the realm of automatic differentiation. It is indeed the core of the process of reverse automatic differentiation, which was introduced in machine learning for algorithmic efficiency reasons. Reverse automatic differentiation consists in propagating differentials by reversing the order in which the functions were primarily computed. For the composition of two functions \( f \) and \( g \), this means that after computing \( a \) and \( f(a) \), one will compute \( Df(a)(g) \) and only after that compute \( Df(a)(g) \circ Da f \). Computationally, this means that the derivative is computed via a continuation-passing
style transformation [39]. This is made clear by looking at the types of these operations. While the usual forward differential of a function \( f : A \to B \) has type \( Df : A \to A \to B \), its reverse differential has type \( \bar{D}f : A \to B^\perp \to A^\perp \), where \( A^\perp \) typically stands for the linear dual of the vector space \( A \). In finite dimensions, those two types are isomorphic but this is not generally the case otherwise.

### 2.4 Linear Substitution

The last ingredient to understand why Dialectica is reverse differentiation is the notion of linear substitution, introduced by Ehrhard and Regnier in their differential \( \lambda \)-calculus [20]. This calculus introduces a new kind of function former \( D t \cdot u \) called the \emph{differential} of \( t \) at point \( u \), which can be typed with the following typing rule.

\[
\frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash D t \cdot u : A \to B}
\]

In addition to the usual \( \beta \)-rule from the \( \lambda \)-calculus, which turns a \( \beta \)-redex into a substitution, the differential \( \lambda \)-calculus contains a rule \( \beta_D \) that turns a differential redex into a linear substitution.

\[
D (\lambda x. t) \cdot u \to_{\beta_D} \lambda x. (\frac{\partial}{\partial x} \cdot u)
\]

Linear substitution is a process that sums over all the linear occurrences of a variable, see Section 4.4 for an in-depth definition. Importantly, linear substitution of application enjoys the following equation.

\[
\frac{\partial(s \cdot t)}{\partial x} \cdot u = \left( \frac{\partial s}{\partial x} \cdot t \right) u + \left( D s \cdot \left( \frac{\partial t}{\partial x} \cdot u \right) \right) u
\]

### 2.5 Dialectica and Linear Logic

Differential linear logic (DiLL) [21, 32] and differential categories [8, 23] express differentiation directly, without relying on the previously exposed notion of linear substitution. In these systems, differentiation as an operation from non-linear proofs (resp. morphisms) to linear proofs (resp. morphisms).

In the case of DiLL, this is achieved by the introduction of new inferences rules which make the following derivation admissible:

\[
\frac{!A \vdash B}{A \vdash !B}
\]

Proofs of \( A \vdash B \) are linear proofs, while proofs of \( !A \vdash B \) are understood as non-linear proofs of \( B \) under the hypothesis \( A \).

In the case of differential categories, differentiation is introduced as a natural transformation \( \hat{d} : !A \to A \).

On the one hand, the Dialectica translation is already known to factor through LL. In this paper, we show that, in fact, it factors through DiLL (Proposition 5.4). More importantly, Dialectica does nothing more than using DiLL rules on LL implications (Proposition 5.1). For this result to go through, the classical nature of DiLL is essential so as to interpret the reverse part of Dialectica by duality.

On the other hand, we show that Dialectica categories over co-Kleisli categories of differential categories readily embed the reverse differentiation functor (Proposition 5.7). This contrasts with traditional approaches where Dialectica categories are seen as an instance of the Chu translation, and used to construct new categorical models of LL [27].

### 2.6 Related work

As far as we know, this is the first time a parallel has been drawn between Dialectica and reverse differentiation. However, in hindsight several lines of work around Dialectica are ominously reminiscent of differentiable programming. Powell [36] formally relates the concept of learning with Dialectica realizers. His definition of learning algorithm is tied to the notion of approximation. Differentiation being just the best linear approximation, our work merely formalizes this relation with linearity. From the categorical point of view, Dialectica is related to lenses [26], which provide themselves a sound categorical interpretation for gradient based learning [14]. More generally, Dialectica is also known to extract quantitative information from proofs [29], which relates very much with the quantitative point of view that differentiation has brought to \( \lambda \)-calculus [5].

### 2.7 Outline

We begin this paper in Section 3 by reviewing the functorial and computational interpretation of differentiation, mainly brought to light by differentiable programming. In particular, we recall Brunel, Mazza and Pagani’s result that reverse differentiation is functorial differentiation where differentials are typed by the linear negation. The most involved section is Section 4, devoted to the computational interpretation of Dialectica. There we recall Pédrot’s sound computational interpretation [34] and the rules of the differential
\(\lambda\)-calculus, to finally show that Pédrot’s reverse translation correspond on first-order terms to a reverse version of the differential \(\lambda\)-calculus linear substitution. In Section 5 we show that the factorization of Dialectica through LL refines to DiLL. Then in Section 5.2 we recall the definition of a Dialectica category and show that it factors through \(\ast\)-autonomous differential categories, which are exactly the models of DiLL. We conclude by some perspective on the possible outcomes of this correspondence.

3 DIFFERENTIABLE PROGRAMMING

We give here an introduction to Automatic Differentiation (AD) oriented towards differential calculus and higher-order functional programming. Our presentation is free from partial derivatives and Jacobians notations, which are traditionally used for presenting AD. We refer to [8] for a more comprehensive introduction to automatic differentiation. We write \(D_t(f)\) for the differential of \(f\) at \(t\). We denote by \(-\cdot-\) the pointwise multiplication of reals or real functions.

Let us recall the chain rule, namely for any two differentiable functions \(f : E \to F\) and \(g : F \to G\) and a point \(t \in E\) we have
\[
D_t(g \circ f) = D_{f(t)}(g) \circ D_t(f).
\]

When computing the value of \(D_t(g \circ f)\) at a point \(v : E\) one must determine in which order the following computations must be performed: \(f(v), D_t(f)(v)\), 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 differentiation amounts to computing first \(f(v)\), then \(g(f(v))\), then the function \(D_{f(v)}(g)\), then computing \(D_t(f)\) and lastly the application of \(D_{f(t)}(g)\) to \(D_t(f)(v)\). Conversely, forward differentiation computes first \(f(v)\), then \(D_t(f)\), then only \(g(f(v))\), then the function \(D_{f(v)}(g)\) and lastly applies \(D_{f(v)}(g)\) to \(D_t(f)\). This explanation naturally fits into our higher-order functional setting. For a diagrammatic interpretation, see for example [10].

These two techniques have different efficiency profiles, depending on the dimension of \(E\) and \(F\) as vector spaces. Reverse differentiation is more efficient for functions with many variables going into spaces of small dimensions. When applied, they feature important optimizations: in particular, differentials are not propagated through higher-order functionals in the chain rule but they are propagated compositionally. The systems we will present in Section 4.1 are not designed with efficiency in mind, and will in particular be completely oblivious to this kind of optimizations. To the risk of repeating ourselves, algorithmic efficiency is not the purpose of this paper, rather we wish to weave equational links between differentiation and Dialectica. As such we do not prove any complexity result.

Brunel, Mazza and Pagani [10] refined the functional presentation by Wang and al. [39] using a linear negation on ground types, and provided complexity results. What we present now is very close in spirit, although their work relies mainly on computational graphs while ours is directed towards type systems and functional analysis. At the core, and as in most of the literature [1, 22, 39], their differential transformation acts on pairs in the linear substitution calculus [2], so as to make it compositional. Consider \(f : \mathbb{R}^n \to \mathbb{R}^m\) differentiable. Then for every \(a \in \mathbb{R}^n\), one has a linear map \(Daf : \mathbb{R}^n \to \mathbb{R}^m\), and the forward differential transformation has type
\[
\overline{D}(f) : (a, x) \in \mathbb{R}^n \times \mathbb{R}^n \mapsto (f(a), Daf \cdot x) \in \mathbb{R}^m \times \mathbb{R}^m
\]

where \(\cdot\cdot\cdot\) stands for the scalar product.

In backward mode, their transformation also acts on pairs, but with a contravariant second component, encoded via a linear dual \((\cdot)^\perp\). The notation \((\cdot)^\perp\) is borrowed from LL, where the (hence linear) negation is interpreted denotationally as the dual on re.

Thus, an element of \(A^\perp\) is a map which computes linearly on \(A\) to return a scalar in \(\mathbb{R}\).
\[
\overline{B}(f) : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m \times \mathbb{R}^n
\]

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 \(f\).

Remark 1. 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: \(f : E \to F\), one has:
\[
\overline{B}(f) : E \times F' \to E \times F'
\]

Consider a function \(g : F \to G\). Then \(\overline{B}(f)\) has the type \(F \times G' \to G \times F'\). If \(G\) and \(F\) are not self-dual, there is no way to define the composition of \(\overline{B}(f)\) with \(\overline{D}(g)\). Thus higher-order differentiation is achieved using two distinct differential transformations. This is the case in the differential \(\lambda\)-calculus for forward AD or the Dialectica Transformation for reverse AD, as we show in Section 4.

However, many functional transformations satisfy the chain rule [3]. Our goal is thus to strengthen our claim and leave no doubt to the reader that Dialectica is indeed differentiation.

4 THE COMPUTATIONAL DIALECTICA AND BACKPROPAGATION

In this section, we tackle what we believe to be the most solid link between Dialectica and Reverse Differentiation. We show that the computational content of Dialectica embeds into a continuation passing-style differential \(\lambda\)-calculus. This is done via logical relations in section 4.5 and via an ad-hoc translation in section 4.6. We first recall the computational presentation of Dialectica in section 4.1, before exhibiting the chain rule in this context in section 4.2 and insisting on its rich type theory in section 4.3.

4.1 An account of the modern Dialectica transformation

After its original presentation by Gödel, Dialectica has been refined as a logical transformation acting from LL to the simply-typed \(\lambda\)-calculus with pairs and sums, by looking at the witness and counter types [16].
This presentation allows removing a lot of historical complexity, including the need for sequences of variables.

In modern terms, one would call it a realizability interpretation over an extended λ-calculus, whose effect is to export intensional content from the underlying terms, i.e. the way variables are used. In its first version however, it relied on the existence of dummy terms at each type and on decidability of the orthogonality condition. The introduction of "abstract multiset" allows getting ride of the decidability condition and makes Dialectica preserve β-equivalence, leading to a kind of Diller-Nahm variant [17].

We recall below the Dialectica translation of the simply-typed λ-calculus. Types of the source language are inductively defined as

\[ A, B \triangleq \alpha \mid A \Rightarrow B \mid A \times B \]

where \( \alpha \) ranges over a fixed set of atomic types. Terms are the usual λ-terms endowed with the standard β, η-equational theory.

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

We furthermore expect that abstract multisets satisfy the following equational theory. Formally, this means that \( \mathcal{M} \) is a monad with a semimodule structure over \( \mathbb{N} \).

**Definition 4.1.** An abstract multiset structure is a parameterized type \( \mathcal{M} (\, - \,) \) equipped with the following primitives.

\[
\begin{align*}
\Gamma \vdash m_1 : \mathcal{M} A & \quad \Gamma \vdash m_2 : \mathcal{M} A \\
\Gamma \vdash t : A & \quad \Gamma \vdash t : A \\
\Gamma \vdash \{ t \} \Rightarrow f & \Rightarrow f t \quad \Gamma \vdash (\lambda x. \{ x \}) \Rightarrow t \\
\Gamma \vdash (t) : \mathcal{M} A & \quad \Gamma \vdash f : \mathcal{M} B \\
\Gamma \vdash m \gg f & \Rightarrow f \mathcal{M} B
\end{align*}
\]

We furthermore expect that abstract multisets satisfy the following equational theory. Formally, this means that \( \mathcal{M} A \) is a monad with a semimodule structure over \( \mathbb{N} \).

**Monadic laws.**

\[
\begin{align*}
\mathcal{W}(\alpha) & \triangleq \alpha \mathcal{W} \\
\mathcal{C}(\alpha) & \triangleq \alpha \mathcal{C} \\
\mathcal{W}(A \Rightarrow B) & \triangleq (\mathcal{W}(A) \Rightarrow \mathcal{W}(B)) \\
\mathcal{C}(A \Rightarrow B) & \triangleq \mathcal{W}(A) \Rightarrow \mathcal{C}(B)
\end{align*}
\]

We now turn to the Dialectica interpretation itself, which is defined at Figure 2, and that we comment hereafter. We need to define the translation for types and terms. For types, we have two translations \( \mathcal{W}(\, - \,) \) and \( \mathcal{C}(\, - \,) \), which correspond to the types of translated terms and stacks respectively. For terms, we also have two translations \( (\, - \,)^* \) and \( (\, - \,)_x \), where \( x \) is a λ-calculus variable from the source language. According to the thesis defended in this paper, we call \( (\, - \,)^* \) the forward transformation, corresponding to the AD forward pass, and \( (\, - \,)_x \) the reverse transformation.

**Theorem 4.2 (Soundness [34]).** If \( \Gamma \vdash t : A \) in the source then we have in the target

- \( \mathcal{W}(\Gamma) \gg (t)_x : \mathcal{M} X \) provided \( x : X \in \Gamma \).

Furthermore, if \( t \equiv u \) then \( t^* \equiv u^* \) and \( t_x \equiv u_x \).

From [34], it follows that the \( (\, - \,)_x \) translation allows observing the uses of \( x \) by the underlying term. Namely, if \( t : A \) depends on some variable \( x : X \), then \( t_x : C(A) \Rightarrow \mathcal{M} C(X) \) applied to some stack \( \pi : \mathcal{C}(A) \) produces the multiset of stacks against which \( x \) appears in head position in the Krivine machine when \( t \) is evaluated against \( \pi \).
\[ W(\mathbb{R}) := \mathbb{R} \quad C(\mathbb{R}) := 1 \]
\[ \varphi^* := (\varphi, \lambda \pi. \alpha \cdot ([()] \mapsto \varphi'(\alpha))) \quad \varphi_x := \lambda \pi. \emptyset \]

Figure 3: Dialectica Derivative Extension

4.2 A differential account of the modern Dialectica transformation

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 Pearlmuter-Siskind untyped translation implementing reverse AD [33]. As such, it allows extracting derivatives in this very general setting.

Lemma 4.3 (Generalized chain rule). Assuming \( t \) is a source function, let us evocatively and locally write \( t' := t^* \cdot 2 \). 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

\[ (f \circ g)' x \equiv \lambda \pi. (f'(g(x)) \pi) \Leftrightarrow (g'(x)) \]

One can recognize this formula as a generalization of the derivative chain rule where the scalar multiplication, or the composition between functions, has been replaced by the monad multiplication.

The abstract multiset is here to formalize the notion of types endowed with a sum, i.e. a commutative monoid structure. By picking a specific instance of abstract multisets, we can formally show that the Dialectica interpretation computes program differentiation.

Definition 4.4. We will instantiate \( \mathcal{M}(\cdot) \) with the free vector space over \( \mathbb{R} \), i.e. inhabitants of \( \mathcal{M} A \) 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

\[ \{ t_1 \mapsto a_1, \ldots, t_n \mapsto a_n \} \]

for the formal sum \( \Sigma_{a < i < n} (a_i \cdot t_i) \) where \( a_i : \mathbb{R} \) 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 \( \lambda \)-calculus with a type of reals \( \mathbb{R} \). We assume furthermore that the source contains functions symbols \( \varphi, \psi, \ldots : \mathbb{R} \rightarrow \mathbb{R} \) 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.

The soundness theorem is then adapted trivially. As a direct consequence of Lemma 4.3 and the observation that for any two \( \alpha, \beta : \mathbb{R} \) we have

\[ \{(\alpha) \mapsto \alpha \times \beta \} \equiv \{(\alpha) \mapsto \alpha \} \Leftrightarrow \lambda \pi. \{(\alpha) \mapsto \beta \} \]

and thus the following theorem.

Theorem 4.5. The following equation holds in the target.

\[ (\varphi_1 \circ \ldots \circ \varphi_n)^* \cdot 2 \alpha (\cdot) \equiv \{(\cdot) \mapsto (\varphi_1 \circ \ldots \circ \varphi_n)(\alpha)\} \]

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

4.3 Higher dimensions

Dialectica also interprets negative pairs, whose translation will be recalled here. Quite amazingly, they allow to straightforwardly provide differentials for arbitrary functions \( \mathbb{R}^n \rightarrow \mathbb{R}^m \).

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

\[ W(A \times B) := W(A) \times W(B) \quad C(A \times B) := C(A) + C(B) \]

Pairs and projections are translated in the obvious way, and their equational theory is preserved, assuming a few commutation lemmas in the target [35].

Writing \( \mathbb{R}^n := \mathbb{R} \times \ldots \times \mathbb{R} \) \( n \) times, we have the isomorphism

\[ C(\mathbb{R}^n) \rightarrow \mathcal{M}(C(\mathbb{R}^m)) \cong \mathbb{R}^{nm}. \]

In particular, up to this isomorphism, Theorem 4.5 can be generalized to arbitrary differentiable functions \( \varphi : \mathbb{R}^n \rightarrow \mathbb{R}^m \), 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 4.6. The Dialectica interpretation systematically computes the total derivative in a higher-order language.

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 [16], the source language can be equipped with inductive types. It can also be adapted to second-order quantification and even dependent types [34]. 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 \( \lambda \)-calculus. The key ingredient of this expressivity is the generalization of scalars to free vector spaces, as \( \mathbb{R} \equiv \mathcal{M} 1 \). The monadic structure of the latter allows handling arbitrary type generalizations. The downside of this approach is that one cannot apply the transformation over itself, in apparent contradiction with what happens for differentiable functions.

4.4 The differential \( \lambda \)-calculus

In this section, we give a quick recap of the syntax of the differential \( \lambda \)-calculus. This will allow us to relate Dialectica with differentials on \( \lambda \)-terms in Sections 4.5 and 4.6.

The differential \( \lambda \)-calculus was introduced by Ehrhard and Regnier [20] as a syntactic account for the mathematical theory of differential calculus. It extends the \( \lambda \)-calculus with a differential application \( D \cdot s \cdot u \) 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 once. It also follows the traditional mathematical intuition, that is head variables—acting as functions—are linear: by definition, one always has
\[(f + g)(x) = f(x) + g(x)\]
while
\[f(x + y) = f(x) + f(y)\]
is an additional property of \(f\). That is, during the whole computation, a linear argument \(u\) should be used only once in \(D \cdot u\). This is why the authors introduced a new reduction rule for this differential application:
\[D (\lambda z. t) \cdot u \rightarrow_{\beta_D} \lambda z. \left(\frac{\partial u}{\partial z} \cdot u\right).\]
The newly introduced \(\frac{\partial u}{\partial z} \cdot u\) is called the linear substitution of \(z\) by \(u\) in \(t\). Just like the usual substitution, linear substitution is a meta-theoretical operation defined by induction on \(t\). Contrarily to the former, it only replaces a single linear occurrence of \(z\) in \(t\). As a result, not all occurrences of \(z\) are replaced at the same time, hence \(z\) may still be free in \(\frac{\partial u}{\partial z} \cdot u\), and thus \(D (\lambda z. t) \cdot u\) reduces to a \(\lambda\)-abstraction that binds \(z\). We now detail the syntax and operational semantics of this calculus.

**Terms of the differential \(\lambda\)-calculus.** We write simple terms as \(s, t, u, v\) while sums of terms are denoted with capital letters \(S, T, U\). The set of simple terms is denoted \(\Lambda\) and the set of sums of terms is denoted \(\Lambda^d\). They are constructed according to the following quotient-inductive syntax.

\[s, t, u, v \quad \in \quad \Lambda^s \quad ::= \quad x \mid \lambda x. s \mid s T \mid D s \cdot t\]

\[S, T, U, V \quad \in \quad \Lambda^d \quad ::= \quad 0 \mid s \mid S + T\]

\[0 + T \equiv T \quad T + 0 \equiv T \quad S + T \equiv T + S\]

We write \(\lambda x. s_i\) for \(\sum_i \lambda x. s_i\cdot (\sum_i s_i) T\) for \(\sum_i s_i T\), and \(D(\sum_i s_i)\cdot (\sum_i j) D s_i \cdot t_j\).

Reduction in the differential \(\lambda\)-calculus is the smallest relation generated by the two rules:

\[(\lambda x. s) T \rightarrow_{\beta} s[x \leftarrow T]\]

\[D (\lambda x. s) \cdot t \rightarrow_{\beta_D} \lambda x. \left(\frac{\partial u}{\partial x} \cdot t\right)\]

and closed by the usual contextual rules.

We consider moreover the simple terms of differential \(\lambda\)-calculus up to \(\eta\)-reduction: in the abstraction \(\lambda x. s\), \(x\) is supposed to be free in \(s\). We denote \(=\) the equivalence relation generated by \(\beta, \beta_D, \eta\) and associativity axioms for +.

The simply-typed \(\lambda\)-calculus can be extended straightforwardly to handle this generalized syntax, in a way which preserves properties such as subject reduction. In particular the differential can be typed by the rule below.

\[
\frac{
\Gamma \vdash s : A \rightarrow B \quad \Gamma \vdash t : A
}{
\Gamma \vdash D s \cdot t : A \rightarrow B
}
\]

**Linear substitution.** We recall the rules of linear substitution in Figure 5. The central and most intricate of them is the one defining linear substitution on an application. It follows the simple fact that a linear variable should be used exactly once. This is illustrated for example in the rule for linearly substituting in an application 10, which we present here in a simpler form.

\[
\frac{
\partial(s u) \cdot t = \left(\frac{\partial u}{\partial x} \cdot t\right) u + D s \cdot \left(\frac{\partial u}{\partial x} \cdot t\right) u
}{
\frac{
\partial(s u) \cdot t = \left(\frac{\partial u}{\partial x} \cdot t\right) u + D s \cdot \left(\frac{\partial u}{\partial x} \cdot t\right) u
}
\]

Figure 5: Linear substitution

If \(z\) is linear in \(s\), then it is in \(s u\). To substitute linearly \(z\) by \(u\) in \(s u\), we can then substitute it linearly in \(s\) and then apply the result to \(u\). But we can also look for a linear occurrence of \(z\) in \(v\). In that case, for \(v\) to remain linear in \(\frac{\partial u}{\partial x} \cdot u\), we should linearize \(s\) before applying it to \(\frac{\partial u}{\partial x} \cdot u\). Then \(s\) will be fed by a linear copy of \(\frac{\partial u}{\partial x} \cdot u\), and then it will be fed by \(u\) as usual. This last case is the computational interpretation for the chain rule in differential calculus.

Types of terms that are used in linear substitution are simpler than in Dialectica.

**Lemma 4.7.** \(\{1, 3, 4\}\) Let \(\Gamma, x : X \vdash t : A\) and \(\Gamma \vdash u : X\). Then

\[
\begin{align*}
\forall x(\Gamma), x : X & \vdash _t : C(A) \Rightarrow \forall x C(X).
\end{align*}
\]

This is particularly obvious when \(t\) is a \(\lambda\) abstraction. While \(\lambda y.s\) is destined to compute on \(y\) before computing on \(x, \lambda z. \frac{\partial y.s}{\partial z} \cdot z\) does the reverse and first waits for \(y\) to be substituted before computing on \(x\).

### 4.5 Relating Dialectica and the differential \(\lambda\)-calculus

In what follows, we show that Dialectica and the differential \(\lambda\)-calculus behave essentially the same by defining a logical relation between those two languages. Actually, since we have two classes of objects, witnesses and counters, we need to define not one but two relations mutually recursively. We will implicitly cast pure \(\lambda\)-terms into the differential \(\lambda\)-calculus.

**Definition 4.8.** Given two simple types \(A\) and \(X\), we mutually define by induction on \(A\) two binary relations

\[
\begin{align*}
\vdash_A & \subseteq \{ t : \Lambda^X \mid \vdash : \mathcal{W}(A) \times \{ T : \Lambda^d \mid \vdash : T : A\} \\
\vdash_A & \subseteq \{ \phi : \Lambda^X \mid \vdash : \mathcal{C}(A) \rightarrow \mathcal{W}(C(X)) \times \{ K : \Lambda^d \mid \vdash : K : X \rightarrow A\} \}
\end{align*}
\]
As is usual, we implicitly close the relation by the equational theory of the corresponding calculus.

- For any atomic type α, we assume given base relations \( \sim_\alpha \) and \( \approx_\alpha \) satisfying further properties specified below.
- The recursive case for arrow types is defined at Figure 6.

In the remainder of this section, we assume that the atomic logical relations satisfy the closure conditions of Figure 7. The first two rules ask for the relation to be compatible with the additive structure of \( \mathcal{M} (\sim_\cdot) \) on the one hand and \( \Lambda^\times \) on the other. In the third rule, \( \Gamma \) stands a list of types and all notations are interpreted pointwise. This rule is asking for the compatibility of the return operation of the multiset monad. We do not need an explicit compatibility with \( \approx_\cdot \) because it will end up being provable in the soundness theorem.

**Lemma 4.9. The closure properties of Figure 7 generalize to any simple type.**

**Theorem 4.10. If \( \Gamma \vdash t : A \) is a simply-typed \( \lambda \)-term, then**

- for all \( r \sim_\cdot \vec{R} \), \( t^* [\Gamma \leftarrow \vec{r}] \sim_\cdot t [\Gamma \leftarrow \vec{R}] \),
- and for all \( \vec{r} \sim_\cdot \vec{R} \) and \( x : X \in \Gamma \),

\[
t_x [\Gamma \leftarrow \vec{r}] \approx_\lambda \lambda z_\cdot \left( \frac{\partial t}{\partial x} \cdot (k \cdot z) \right) [\Gamma \leftarrow \vec{R}].
\]

**Proof.** As usual, the proof goes by induction over the typing derivation. We need to slightly strengthen the induction hypothesis by proving a generalized form of substitution lemma relating \( \approx_\cdot \) on the left with composition on the right, i.e. for any \( \phi \approx_\lambda k \) then

\[
(\lambda \pi. t_x [\Gamma \leftarrow \vec{r}] \pi \approx_\lambda \phi) \approx_\lambda \lambda z_\cdot \left( \frac{\partial t}{\partial x} \cdot (k \cdot z) \right) [\Gamma \leftarrow \vec{R}]
\]

from which the second statement of the theorem is obtained by picking \( \phi : \lambda \pi. \{x\} \) and \( k : \lambda z \cdot z \), which are always in relation by Lemma 4.9. The proof is otherwise straightforwardly achieved by equational reasoning. \( \square \)

This theorem is a formal way to state that the Dialectica interpretation and the differential \( \lambda \)-calculus are computing the same thing without having to embed them in the same language. It makes obvious the relationship between the \( (\cdot)_x \) interpretation and the \( \frac{\partial t}{\partial x} \cdot (\cdot) \) operation. Interestingly, \( \approx_\lambda \cdot \cdot \approx_\lambda \) relates two functions going in the opposite direction. While the left-hand side has type \( \mathbb{C}(A) \to \mathcal{M} \mathbb{C}(X) \) in \( \lambda^\times \), the right-hand side has type \( X \to A \) in the differential \( \lambda \)-calculus. We believe that this is a reflection of the isomorphism between a linear arrow and its linear contrapositive, since both sides of the relation are actually linear functions.

**Remark 2.** This distinction in Pédrot’s Dialectica between summable and non-summable terms strongly relates with Erhard’s recent work on deterministic probabilistic coherent spaces [19].

### 4.6 Translating Dialectica into differential \( \lambda \)-calculus

In this section, we show that the two transformation acting on \( \lambda \)-terms in Dialectica are a CPS version of the ones in differential \( \lambda \)-calculus. We define a translation on counter terms to make the previous logical relation a translation from Dialectica terms to differential \( \lambda \)-calculus. This translation works basically as the logical relation before, but with an enforced CPS translation to retrieve forward differentiation.

**Definition 4.11.** Consider a term \( s \) of the \( \lambda^\times \)-calculus, typed in some context \( \Gamma \) by \( \Gamma \vdash s : S \). Define \( [s] \) as follows:

- If \( S = A \times B \) is pair, then \( [s] : = \lambda k . ([s][,2](k(s_1))) \),
- Otherwise \( [s] := \lambda k . k s \).

**Lemma 4.12.** If \( s \equiv s' \) then \( [s] \equiv [s'] \).

This translation is directed by the intuition that a term typed by a counter type \( s : \mathbb{C}(A) \) can be translated to a term typed by a linear dual to a witness type \( \mathcal{W}(A) \) through the rules of Linear Logic (see section 5 later). Taking into account the involutivity of duals in Linear Logic, we have thus for a term \( s \) of type \( \mathbb{C}(A) \Rightarrow B = \mathcal{W}(A) \times \mathbb{C}(B) \):

\[
[s] : \mathcal{W}(A) \times \mathcal{W}(B)^\perp \equiv (\mathcal{W}(A) \times \mathcal{W}(B)^\perp)^\perp \equiv (\mathcal{W}(A) \Rightarrow \mathcal{W}(B))^\perp \equiv (\mathcal{W}(A) \Rightarrow B).1^\perp
\]

For \( s \) a term of the \( \lambda^\times \)-calculus we make \( [\cdot] \) distribute over sums and translate \( s \) into a term of the differential \( \lambda \)-calculus:

\[
\begin{align*}
[0] & : = \lambda k . k 0 \\
[t \# u] & : = [t] + [u] \\
[t] & : = [t].
\end{align*}
\]

**Theorem 4.13.** Consider \( t \) a simply-typed \( \lambda \)-term, a term of the \( \lambda^\times \)-calculus \( u \), and a variable \( x \) such that in some context \( \Gamma \) we have \( \Gamma ; x : X \vdash t : B \) and \( \Gamma \vdash u : \mathcal{M} \mathbb{C}(B). \) Then

\[
[u \Rightarrow t_x [\Gamma \leftarrow \vec{r}]] \Rightarrow_{\beta_\lambda} \lambda z_\cdot \left( \frac{\partial t}{\partial x} \cdot (k \cdot z) \right)
\]

The proof goes by induction on the typing derivation of \( t \), for any variable \( x \) and counter witness \( u \). It makes a heavy use of the monadic and monoidal laws on \( \mathcal{M} (\cdot) \) recalled in Section 4.1, and of the fact that the linearity of witnesses is now enforced. The proof is detailed in the appendix.
5 DIALECTICA FROM LL TO DiLL

The close relationship between Dialectica and Linear Logic was identified as the second was found. Dialectica factorizes through Linear Logic and transports its structure to new models for it. In this section, we show that more precisely, Dialectica merely adds rules from Differential Linear Logic to Linear Logic, and embeds faithfully reverse functorial differentiation in its arrows. We do that in sequent calculus in section 5.1 and in categorical models of Dialectica in 5.2.

5.1 The linear Dialectica is differential

In this section, we show that Dialectica factorizes through DiLL, and that rules of DiLL prove Dialectica.

Linear Logic. Formulas of LL are constructed according to the following grammar.

\[ A, B : = 0 | 1 | \bot | \top | A \otimes B | A & B | A \multimap B | A \otimes B | A \otimes B | A ! | ?A \]

We define as usual the involutive negation \((-)^{\bot}\), & being the dual of \(@\), \(\oplus\) the dual of \(\otimes\) and \(!\) the dual of \(?\). As per the standard practice, we define the linear implication \(A \multimap B : = A^{\bot} \otimes B\), from which the usual non-linear implication can be derived through the call-by-name encoding \(A \Rightarrow B : = !A \Rightarrow B\), where the exponential formula \(!A\) represents the possibility to use \(A\) an arbitrary number of times.

Dialectica through Linear Logic. The Dialectica translation can in fact be modernized as a translation to and from types of \(\lambda^{+,\times}\)-calculus: this was already described on implications in Figure 2 and is fully recalled in Figure 8.

This translation factorizes as a translation LL connectives into intuitionistic types [16], described in Figure 9. It factorizes through the linear Dialectica by injecting Lj into LL via the economical

\[ W(0) : = 1 \quad C(0) : = 1 \quad W(1) : = 1 \quad C(1) : = 1 \]
\[ W(A \times B) : = W(A) \times W(B) \quad C(A \times B) : = C(A) \times C(B) \]
\[ W(A + B) : = W(A) + W(B) \quad C(A + B) : = C(A) \times C(B) \]
\[ C(A \Rightarrow B) : = (W(A) \Rightarrow W(B)) \times (W(A) \Rightarrow C(A)) \]

Figure 8: A modernized Dialectica [35, 8.3.1]

We now present the Dialectica translation from LL to DiLL in Figure 10. This translation hardwires the fact that an implication must be accompanied by its reverse differential. If the implication depends on an exponential, then some real differentiation will happen, otherwise the translation is straightforward.

Through the usual encoding \(A \Rightarrow B : = A^{\bot} \otimes B\), one has

\[ W(!A \Rightarrow B) = (C(B) \Rightarrow !W(A) \Rightarrow C(A)) \]

As such, the functional translation from LL to DiLL only encodes the differential part of Dialectica.

Proposition 5.1. For any formula \(A\), one has \(A \vdash W(A)\) and \(C(A)^{\bot} \vdash A\). The proof themselves are functional, only in the case \(C(A)^{\bot} \vdash !A\), which uses the coderecursion and co-contraction rule of DiLL.

Proof. We use the fact that, when it is defined, \(W(A^{\bot}) = C(A)\) and show that the statement holds for any connective of LL including \(\otimes\) and \(?\). The proof is then a straightforward induction on the formula \(A\) for any context \(\Gamma\). The only interesting case is the one for the witness to the exponential ?, namely that when \(\Gamma \vdash \exists?A\) then
\[\begin{align*}
\varpi(1) & := 1 \\
\varpi(T) & := T \\
\varpi(A \otimes B) & := \varpi(A) \otimes \varpi(B) \\
\varpi(A^+) & := \varpi(A) \\
\neg \varpi(A) & := \neg \varpi(A)
\end{align*}\]

\[\begin{align*}
\Gamma \vdash \varpi(\alpha) & \equiv \varpi(A^+) \equiv \neg \varpi(A) \varpi(A). \text{The fact that when } \Gamma \vdash \alpha \text{ in LL then } \Gamma \vdash \alpha \varpi \alpha \text{ in DiLL constitute the very heart of Differential Linear Logic, and uses the newly introduced codereliction and cocontraction rules. As LL is a subsystem of DiLL, from a proof } \pi \text{ of } \Gamma \vdash \alpha \text{ one easily constructs a proof of } \Gamma \vdash \alpha \varpi \alpha \text{ from a derivation on } A^+ \text{ (corresponding to the reverse argument) and a co-contraction on } !A^+ \text{ with an axiom introducing the non-linear argument.}
\end{align*}\]

\[\frac{\vdash A, A^+ \quad \pi}{\vdash \alpha, A, A^+} \quad \text{ax} \]

\[\frac{\vdash \alpha, A, A^+ \quad \varpi}{\vdash \alpha, A, A^+} \quad \text{cut} \]

\[\Gamma \vdash \alpha \quad \varpi \]

\[\square\]

\textbf{Factorisation of Dialectica.} While differential, the translation presented in Figure 11 is not specifically reverse. Indeed, as Differential Linear Logic is classical, one has equivalently:

\[\left(\forall \varpi(A) \rightarrow \neg \varpi(B) \rightarrow \varpi(C)\right) \equiv \left(\forall \varpi(A) \rightarrow \neg \varpi(B) \rightarrow \varpi(C)\right)\]

That is, due to the presence and associativity of the \(\varpi\), or equivalently due to the involutive linear negation reverse and forward derivative are equivalent.

Hence, to recover a factorization of Dialectica through LL and DiLL we must distinguish forward and reverse differentiation formulas in order to force the backward translation. We now make the Dialectica translation act on formulas of intuitionistic LL:

\[A, B := 0 | 1 | T | A \otimes B | A \& B | A \rightarrow B | A \otimes B | !A\]

Figure 12 presents an intuitionistic variant the Dialectica translation from LL to DiLL, varying from figure 12 only on multiplicative connectives.

To recover the translations from and to types of \(\lambda^{+,\times}\)-calculus through the types of \(\lambda\)-calculus, we refine the economical translation. We interpret the arrow by both a call-by-name arrow, which will be differentiated, and a linear arrow, which will be translated itself after going through DiLL.

\textbf{Definition 5.2.} The following defines a translation from types of \(\lambda^{+,\times}\) to LL:

\[\begin{align*}
[A \times B]_d & := A \& B \\
[A + B]_d & := A \oplus B \\
[A \Rightarrow B]_d & := (\neg A \Rightarrow B) \& (A \Rightarrow B) \\
[0]_d & := 0 \\
[1]_d & := 1
\end{align*}\]

\textbf{Definition 5.3.} The translation from intuitionistic DiLL to types of \(\lambda^{+,\times}\)-calculus is defined as follows:

\[\begin{align*}
\mathcal{U}(!A) & := A \\
\mathcal{U}(A \& B) & := \mathcal{U}(A) \times \mathcal{U}(B) \\
\mathcal{U}(A + B) & := \mathcal{U}(A) + \mathcal{U}(B) \\
\mathcal{U}(A \Rightarrow B) & := \mathcal{U}(A) \Rightarrow \mathcal{U}(B) \\
\mathcal{U}(1) & := \mathcal{U}(\emptyset) := 1
\end{align*}\]

We then obtain the expected commutative diagram. The proof proceeds by an immediate induction on the syntax of formulas. Note that we used the same notation for witness and counter types of LL and \(\lambda^{+,\times}\), but they can easily be discriminated from the context. The counter witness for implication is only recovered up to logical equivalence, as \(\mathcal{U}(\mathcal{C}_i(A \Rightarrow B)) \equiv (\mathcal{U}(\mathcal{C}_i(A)) \times \mathcal{U}(\mathcal{C}_i(B))) \oplus (\mathcal{U}(\mathcal{C}_i(A)) \times \mathcal{U}(\mathcal{C}_i(B)))\).

\textbf{Proposition 5.4.} The Dialectica transformation on types factorizes through LL and DiLL as follows:

\[\begin{array}{c}
\text{LL} \\
\llw \downarrow \mathcal{U} \\
\Gamma \Rightarrow \mathcal{C}_i \\
d \text{DiLL} \\
\llw \uparrow \mathcal{U}
\end{array}\]

The advantage of this translation is that it isolate the duplication of function to the \([\_\_\_\_]\) translation, while the purely differential part is embedded into a translation from LL to DiLL. Note that the translation of \(\mathcal{C}_i(A \& B)\) as \((\mathcal{U}(\mathcal{C}_i(A)) \& \mathcal{U}(\mathcal{C}_i(B)))\) is necessary only for the decomposition of witness and counter types from LL to \(\lambda^{+,\times}\) through DiLL. If one does not try to decompose this arrow and insert it as a diagonal in the diagram above, then one could have interpreted \(\mathcal{C}_i(A \& B)\) as \((\mathcal{U}(\mathcal{C}_i(A)) \& \mathcal{U}(\mathcal{C}_i(B)))\).

\[\begin{array}{c}
5.2 \text{ Dialectica and Differential Categories} \\
\text{This section finally tackles the semantical side of the correspondence between Dialectica and Differentation. We show that Dialectica constructions over the co-Kleisli of differential categories faithfully embed functorial reverse differentials.}
\end{array}\]

The Dialectica transformation was studied from a categorical point of view by De Paiva and Hyland [16]. They have been used as a way to generate new models of LL [16, 26]. Our point of view is quite orthogonal. We prove that they also characterize specific models of LL: if \(C\) is a model of LL, then the Dialectica Category constructed on \(C\) inherits its monoidal and exponential structure, to make it a new model of LL.

\textbf{Definition 5.5} ([16]). Consider \(C\) a category with finite limits. The Dialectica category \(\mathcal{D}(C)\) over \(C\) has as objects relations \(\alpha \subseteq (A, X)\) on objects of \(C\), and as arrows pairs \((f, F) : \alpha \subseteq (A, X) \rightarrow \beta \subseteq (B, Y)\) of maps

\[\begin{array}{c}
f : A \rightarrow B \\
F : A \times Y \rightarrow X
\end{array}\]

such that if \((a; F(a); y) \in \beta\) then \((f(a); y) \in \beta\). Consider

\[\begin{align*}
(f, F) : \alpha & \subseteq (A, X) \\
(\gamma, G) : \beta & \subseteq (B, Y)
\end{align*}\]

and

\[\alpha \subseteq (A, X) \quad \beta \subseteq (B, Y) \quad \gamma \subseteq (C, Z)\]
two arrows of the Dialectica category. Then their composition is defined as
\[
(g, G) \circ (f, F) := (g \circ f, (a, z) \mapsto F(a, G(f(a), z)).
\]

The identity on an object \( \alpha \subseteq (A, X) \) is the pair \((id_A, (\_).2)\) where \((\_).2\) is the projection on the second component of \(A \times X\).

In our point of view, objects of \( \mathcal{D}(C) \) generalize the relation between a space \( A \) and its tangent space. Arrows \((f, F)\) represent a function and its reverse differential \( f \), according to the typing intuition developed in Section 3. Composition is exactly the chain rule.

Categorical axiomatizations of differentiation has been widely studied as stemming from models of Differential Linear Logic. The various axiomatizations such as differential, cartesian differential or tangent categories \([8, 9, 12]\), all encode forward derivatives. To generalize the relation \( d^k \in \mathcal{L}(\mathcal{L}(B^2), B^2) \) and the strong monoidality of \( ! \), one gets a morphism:

\[
\overrightarrow{D}(f) \in \mathcal{L}(!(A \times B^1), A^2).
\]

**Proposition 5.7.** In the setting described above, one has a functor from the co-Kleisli \( \mathcal{L}_1 \) to the Dialectica category over it \( \mathcal{D}(\mathcal{L}_1) \):

\[
\begin{aligned}
\mathcal{L}_1 &\quad \to \quad \mathcal{D}(\mathcal{L}_1) \\
A &\quad \mapsto \quad A \times A^1 \\
f &\quad \mapsto \quad (f, \overrightarrow{D}(f))
\end{aligned}
\]

**Proof.** If \( f \) is a morphism from \( A \to B \) in \( \mathcal{L}_1 \), then \( f \in \mathcal{L}(A, B) \) and \( \overrightarrow{D}(f) \) is a morphism from \( A \times B^1 \) to \( A^1 \) in \( \mathcal{L}_1 \). Therefore, if \( f \) is the identity on \( A \) in \( \mathcal{L}_1 \), and that is \( d = d_A \in \mathcal{L}(A, A) \), then the comonoid equation for \( d \) [23, Definition 4.2.2] ensures that \( \overrightarrow{D}(f) \) is indeed the projection on the second component. Finally, if \( f \in \mathcal{L}(A, B) \) and \( g \in \mathcal{L}(B, C) \), then the second monad rule guarantees that

\[
g \circ f \circ \mu \circ d = g \circ d \circ (f \circ \partial \circ !) \circ (1 \otimes \overrightarrow{m})
\]

where \( \overrightarrow{m} \) is the composition of the biproduct diagonal and the comonad strong monoidality. See the literature [23] for explicit handling of annihilation operators and coproducts in this formula, which is nothing but the categorical restatement of the chain rule. The above formula then exactly corresponds to the composition in \( \mathcal{L}_1 \) of \( \overrightarrow{D}(g) \) and \( (f \circ \pi_1 \circ \overrightarrow{D}(f)) \), modulo the strong monoidality of \( ! \). \( \square \)

We have an immediate forgetful functor:

\[
\Pi_1 : \quad \mathcal{D}(\mathcal{L}_1) \quad \to \quad \mathcal{L}_1
\]

\[
\begin{aligned}
\mathcal{D}(\mathcal{L}_1) &\quad \to \quad \mathcal{L}_1 \\
A &\quad \mapsto \quad A \\
f, F &\quad \mapsto \quad f
\end{aligned}
\]

However, it does not result in an adjunction between \( \mathcal{L}_1 \) and \( \mathcal{D}(\mathcal{L}_1) \) as without any linearity condition on \( F \), it might not be equal to its own reverse derivative. With a linearity condition however, unicity of differentiation holds [30], and an adjunction between \( \Pi_1 \) and \( \overrightarrow{D} \) should be ensured.

**Reverse derivative categories.** This setting can surely be relaxed, and might be broader relations between Dialectica categories and differential categories. In particular, if \( C \) is a reverse derivative category [13], one should construct a functor

\[
\begin{aligned}
C &\quad \to \quad \mathcal{D}(C) \\
A &\quad \mapsto \quad A \times A \\
f &\quad \mapsto \quad (f, R[f])
\end{aligned}
\]
where $R[f]$ represents the reverse derivative of an arrow $f$ as described in the paper.

6 CONCLUSION AND PERSPECTIVES

In this paper we related the different interpretations of Gödel’s Dialectica with logical differentiation. We first studied Dialectica as a transformation on $\lambda$-terms, and showed that it corresponds to a reverse differential $\lambda$-calculus. We then explored how Dialectica consists in adding rules of DiLL to connectives of LL, and embedded differential categories into Dialectica Categories. The absence of formal link between Section 4.1 and 5.1 could be worrying to the reader: why is one not the direct translation of the other? The reason however is that differential $\lambda$-calculus is not typed by DiLL. While differential $\lambda$-calculus has the same models [] and originate from the same structure [18] as DiLL, they are not related with the Curry-Howard correspondence. We are missing a proof-term language for DiLL: below we suggest a few automatic differentiation features that could come from it. Finally, this paper overlooked an essential feature of Dialectica: its main use today is in so-called Applied Proof Theory, as it allows extracting quantitative statements from existence theorem in mathematics. Deepening the connection between differentiation and refinement of mathematical theorems seems an exciting task to us.

Automatic differentiation and reduction strategies. The Dialectica interpretation explored in this paper is fundamentally call-by-name on the arrow, as recalled in section 5 or in its categorical semantics. This points out that the call-by-name interpretation of functions and their derivative might implement some kind of reverse derivative. The consequences of this could be interesting in a language typed by Differential Linear Logic. Indeed, in the semantics of Differential Linear Logic, non-linear functions $f$ are are seen as functions $\tilde{f}$ that act on distributions [38] [28]. These comes as traditional arguments, encoded through diracs:

$$\tilde{f}(\delta_a) \rightarrow f(a),$$

or they act on differentiated arguments

$$\tilde{f}(D_0(\_a)) \rightarrow D_0(f)a.$$

Giving the priority to the evaluation of $f$ (call-by-name) relate to backward differentiation, while giving the priority to $D_0(\_a)$ (call-by-value) relates to forward differentiation. An exploration to the $L$-calculus [15] adapted to Differential Linear Logic and linear context could allow expressing such principles.

Proof mining and differentiation. Proof mining [29] consists in applying logical transformations to mathematical proofs, in order to extract more information from these proofs and refine the theorem they prove. This has been particularly effective in functional analysis, where logicians are able to transform existential proofs into quantitative proofs. For instance, from unicity proofs in approximation theory one gets an effective moduli of uniqueness, that is a characterization of the rate of convergence of approximants towards the best approximation.

While metatheorems in proof mining guarantee the existence of constructive proofs, applying the Dialectica transformation to proofs might consists in functional analysis in differentiating the “$\varepsilon$” function. For example, if a unicity statement “$\forall \varepsilon, \exists \eta, |G_\varepsilon (a, b)| < \eta \Rightarrow |a - b| < \varepsilon$” is established, extracting a quantitative rate of convergence would consists in differentiating the function $\varepsilon \mapsto \eta$. Exploring the consequences of metatheorems in proof mining over logical differentiation seems like an interesting perspective.
REFERENCES


A PROOFS

Proof. Proof of Theorem 4.10. As usual, the proof goes by induction over the typing derivation. We need to slightly strengthen the induction hypothesis by proving a generalized form of substitution lemma relating ≫ on the left with composition on the right, i.e. for any φ \equiv \lambda x k then

\[(\lambda \pi \cdot t s (\Gamma \leftarrow \pi) \pi \gg \phi) \equiv \lambda x \lambda z. \left(\frac{\partial t}{\partial x} (k \cdot z)\right) (\Gamma \leftarrow \pi)\]

from which the second statement of the theorem is obtained by picking φ := \lambda x (\pi) and k := \lambda x z, which are always in relation by Lemma 4.9. The proof is otherwise straightforwardly achieved by equational reasoning.

Proof. Proof of Theorem 4.13 Let us prove the result by induction on the typing derivation of t, for any variable x and counter witness w. We will make a heavy use of the monadic and monoidal laws on \(\mathcal{M}(\_\_\_\_)\) recalled in Section 4.1. As the operator \(\otimes\) is associative and commutative, we will denote \(\otimes u\) the term \(u \otimes \ldots \otimes u\) for \(n \in \mathbb{N}^+\). For readability issues, when the substitution \(\Gamma \leftarrow \pi\) does not play a role in the proof we will omit it.

- The variable case is straightforward. If \(t \equiv x\), then as \(u\) is a term of pure \(\lambda\)-calculus we have the monads:

\[\llbracket u \gg x_s \rrbracket = \llbracket u \gg \lambda x (\pi) \rrbracket \equiv_\beta \llbracket u \rrbracket\]

On the other hand, by \(\eta\)-equivalence we have

\[\lambda z. \llbracket u \rrbracket \left(\frac{\partial z}{\partial x} \cdot z\right) = (\lambda z. \llbracket u \rrbracket z) \equiv \llbracket u \rrbracket\]

If \(t\) equals another variable \(y \neq x\), then similarly through the distributivity laws:

\[\llbracket u \gg y_s \rrbracket \equiv_\beta \llbracket u \gg \lambda x (\pi) \rrbracket = 0\]

and through the laws of differential \(\lambda\)-calculus [20]

\[\lambda z. \llbracket u \rrbracket \left(\frac{\partial y}{\partial x} \cdot z\right) = \lambda z. \llbracket u \rrbracket 0 \equiv_\beta \lambda z. (0u) \equiv \lambda z. 0 \equiv 0\]

- Let us tackle the abstraction case and suppose

\[t \equiv \lambda y.s : A \Rightarrow B\]

for some variable \(y \neq x\) and some \(\lambda\)-calculus term s. This case is more intricate as, while \((\lambda y.s)\_x\) is destined to be computed on \(x\) before computing on \(y\), \(\frac{\partial (\lambda y.s)}{\partial x} \cdot t\) first linearly substitutes \(x\) with \(t\) before computing on \(y\).

As \(u\) has type \(\mathcal{M}(\mathcal{C}(A \Rightarrow B)) = \mathcal{M}((\mathcal{C}(A) \times \mathcal{C}(B)))\), \(u\) reduces to a possibly empty sum \(\bigoplus_i \{u_i\}\) with \(u_i : \mathcal{C}(A) \times \mathcal{C}(B)\) in the context \(\Gamma\). Indeed, \(\mathcal{M}\) is a monad over the types of the \(\lambda\)-calculus, and as such one does not construct pairs of \(\mathcal{M}\) types. By the definition of \((\lambda y.s)_x\), by application of monadic laws and as \(x\) is free in \(u\):

\[\bigoplus_i \{u_i\} \gg ((\lambda y.s)_x) \llbracket u_i \rrbracket = \bigoplus_i \{\bigoplus_j \{u_i\} \gg ((\lambda y.s)_x) \llbracket u_j \rrbracket\} = \bigoplus_i \{\bigoplus_j \{u_i \cdot 1/y\} \cdot u_j \rrbracket\} = \bigoplus_i \{\bigoplus_j \{u_i \cdot 2\} \gg ((s[\cdot 1/y])_x)\}\]

On the differential \(\lambda\)-calculus side:

\[
\lambda z. \llbracket u \rrbracket \left(\frac{\partial \lambda y.s}{\partial x} \cdot z\right) = \lambda z. \left(\sum_i \llbracket u_i \rrbracket \left(\frac{\partial \lambda y.s}{\partial x} \cdot z\right)\right) = \lambda z. \left(\sum_i \lambda z. (\lambda k. \llbracket u_i \cdot 2\rrbracket (k(u_i \cdot 1))) \left(\frac{\partial \lambda y.s}{\partial x} \cdot z\right)\right).
\]

As \(\frac{\partial \lambda y.s}{\partial x} \cdot z = \lambda y. \frac{\partial s}{\partial x} \cdot z\), we have:

\[
\lambda z. \llbracket u \rrbracket \left(\frac{\partial \lambda y.s}{\partial x} \cdot z\right) = \lambda z. \left(\sum_i \lambda z. \llbracket u_i \cdot 2\rrbracket (\lambda y. \left(\frac{\partial s}{\partial x} \cdot z\right) (u_i \cdot 1)) \right) = \lambda z. \left(\sum_i \lambda z. \llbracket u_i \cdot 2\rrbracket \left(\frac{\partial s}{\partial x} \cdot z\right) (u_i \cdot 1/y)\right).
\]

As \(x\) is free in \(u\), one has

\[
\frac{\partial s}{\partial x} (u_i \cdot 1/y) = \left(\frac{\partial s}{\partial x} (u_i \cdot 1)\right) / y\]

and the induction hypothesis concludes the case:

\[
\lambda z. \left(\sum_i \lambda z. \llbracket u_i \cdot 2\rrbracket \left(\frac{\partial s}{\partial x} \cdot z\right) (u_i \cdot 1/y)\right).
\]

Let us study the application case for the reverse transformation. Suppose that \(t = (s \cdot y)\), where \(s : A \Rightarrow B\) and \(v : A\). Then

\[\llbracket u \gg ((s \cdot v)_x) (\Gamma \leftarrow \pi) \rrbracket = \llbracket u \gg (\lambda \pi. (s \cdot x) \otimes (\pi \cdot y)) \rrbracket \]

We have \(s \equiv \lambda y.s'\) for some term \(s'\) of the pure \(\lambda\)-calculus. As such, \(s' \equiv \lambda y.\lambda y.s'\pi\) and \(s' \equiv \lambda y. (\lambda y.s'\pi)\pi 2.1\) by the soundness theorem 4.2. Thus, thanks to the distributivity laws (Definition 4.1):

\[\llbracket u \gg ((s \cdot v)_x) (\Gamma \leftarrow \pi) \rrbracket = \llbracket u \gg ((\lambda \pi. (\lambda y.s'\pi)\pi) (\Gamma \leftarrow \pi) \rrbracket\]

As \(x\) is free in \(u\), and by induction hypothesis on \(s'\):

\[\lambda z. \llbracket u \rrbracket \left(\frac{\partial s'}{\partial x} (\Gamma \leftarrow \pi) \cdot z\right) = \lambda z. \llbracket u \rrbracket \left(\frac{\partial (\lambda y.s')}{\partial x} (\Gamma \leftarrow \pi) \cdot z\right)\]

As

\[\lambda z. \llbracket u \rrbracket \left(\frac{\partial s'}{\partial x} (\Gamma \leftarrow \pi) \cdot z\right) = \lambda z. \llbracket u \rrbracket \left(\frac{\partial (\lambda y.s')}{\partial x} (\Gamma \leftarrow \pi) \cdot z\right)\]
By the monadic laws on $\rhd$ and by the induction hypothesis on $v$, we have:

\[ \left[ u \rhd \pi. ((y \pi)[\Gamma, y \leftarrow \text{r}^*]) \right] \equiv \pi_x[\Gamma \leftarrow \text{r}^*] \]

\[ = \left[ (u \rhd \pi. ((y \pi)[\Gamma, y \leftarrow \text{r}^*]) \right] \equiv \pi_x[\Gamma \leftarrow \text{r}^*] \]

\[ = \lambda z.[(u \rhd \pi. ((y \pi)[\Gamma, y \leftarrow \text{r}^*, v^*]) \left( \frac{\delta \pi[\Gamma \leftarrow \text{r}^*]}{\delta x} \cdot z \right) \]

By induction hypothesis on $s'$ we then have:

\[ \left[ u \rhd \pi. ((y \pi)[\Gamma, y \leftarrow \text{r}^*]) \right] \equiv \pi_x[\Gamma \leftarrow \text{r}^*] \]

\[ = \lambda z.[u] \left( \frac{\delta ^* \pi'[\Gamma, y \leftarrow \text{r}^*, v]}{\delta y} \cdot z \right) \left[ \delta \pi[\Gamma \leftarrow \text{r}^*, v] \cdot z \right] \]

\[ = \lambda z.[u] \left( \lambda y. \frac{\delta ^* \pi'[\Gamma \leftarrow \text{r}^*, v]}{\delta y} \cdot \frac{\delta \pi[\Gamma \leftarrow \text{r}^*, v]}{\delta x} \cdot z \right) \]

We conclude the case by computing the other hand of the equation:

\[ \lambda z. [u] \left( \frac{\delta (s)u[\Gamma \leftarrow \pi]}{\delta x} \cdot z \right) \]

\[ = \lambda z. [u] \left( \frac{\delta s[\Gamma \leftarrow \pi]}{\delta x} \cdot z \right) v \]

\[ + (D \cdot \left( \frac{\delta \pi}{\delta x} \cdot z[\Gamma \leftarrow \pi] \right) v) \]

\[ = \lambda z. [u] \left( \left( \frac{\delta s[\Gamma \leftarrow \pi]}{\delta x} \cdot z \right) v \right) \]

\[ + (\lambda y. \frac{\delta \pi[\Gamma \leftarrow \pi]}{\delta y} \cdot \frac{\delta \pi[\Gamma \leftarrow \pi]}{\delta x} \cdot z \right) v) \]

\[ = \lambda z. [u] \left( \left( \frac{\delta \pi[\Gamma \leftarrow \pi]}{\delta x} \cdot z \right) v \right) + \left( \lambda y. \frac{\delta \pi[\Gamma \leftarrow \pi]}{\delta y} \cdot \frac{\delta \pi[\Gamma \leftarrow \pi]}{\delta x} \cdot z \right) v) \]

\[ \left[ \Gamma \leftarrow \text{r}^* \right] \]

\[ \square \]