∂ is for Dialectica

Marie Kerjean

CNRS & LIPN, Université Sorbonne Paris Nord

Work in collaboration with Pierre-Marie Pédrot
Gödel’s Dialectica Transformation

- Gödel Dialectica transformation [1958]: a translation from intuitionistic arithmetic to a finite type extension of primitive recursive arithmetic.

\[ A \leadsto \exists u : \mathbb{W}(A), \forall x : \mathbb{C}(A), A^D[u, x] \]

- De Paiva [1991]: the linearized Dialectica translation operates on Linear Logic (types) and \( \lambda \)-calculus (terms).

- Pedrot [2014] A computational Dialectica translation preserving \( \beta \)-equivalence, via the introduction of an ”abstract multiset constructor” on types on the target.
Gödel’s Dialectica

1. \((F \land G)' = (\exists yv) (zw) [A (y, z, x) \land B (v, w, u)]\).
2. \((F \lor G)' = (\exists yvt) (zw) [t=0 \land A (y, z, x) \lor t=1 \land B (v, w, u)]\).
3. \([(s) F]' = (\exists Y) (sz) A (Y (s), z, x)\).
4. \([(\exists s) F]' = (\exists sy) (z) A (y, z, x)\).
5. \((F \supset G)' = (\exists VZ) (yw) [A (y, Z (yw), x) \supset B (V (y), w, u)]\).
6. \((\neg F)' = (\exists \bar{Z}) (y) \neg A (y, \bar{Z} (y), x)\).

Gödel’s Dialectica

- Validates semi-classical axioms:
  - Markov’s principle: \(\neg\neg\exists x A \rightarrow \exists x A\) when \(A\) is decidable.
  - Independant of premises: \((A \rightarrow \exists x B) \rightarrow (\exists x.(A \rightarrow B))\)

- Numerous applications:
  - Soundness results
  - Proof mining

A further distinguishing feature of the D-interpretation is its nice behavior with respect to modus ponens. In contrast to cut-elimination, which entails a global (and computationally infeasible) transformation of proofs, the D-interpretation extracts constructive information through a purely local procedure: when proofs of \(\varphi\) and \(\varphi \rightarrow \psi\) are combined to yield a proof of \(\psi\), witnessing terms for the antecedents of this last inference are combined to yield a witnessing term for the conclusion. As a result of this modularity, the interpretation of a theorem can be readily obtained from the interpretations of the lemmata used in its proof.

A peek into Dialectica interpretation of functions

\[(A \rightarrow B)_D = \exists f g \forall xy (A_D (x, gxy) \rightarrow B_D (fx, y))\]

**Usual explanation**: least unconstructive prenexation.
- Start from \(\exists x, \forall u, A_D [x, u] \rightarrow \exists y, \forall v, B_D [y, v]\).
- Obvious prenexation: \(\forall x (\forall u, A_D [x, u] \rightarrow \exists y, \forall v, B_D [y, v])\)
- Weak form of IP: \(\forall x \exists y, \forall v, \exists u (A_D [x, u] \rightarrow B_D [y, v])\).
- Prenexation: \(\forall x \exists y, \forall v, \exists u (A_D [x, u] \rightarrow B_D [y, v])\).
- Markov: \(\forall x, \exists y, \forall v, \exists u (A_D [x, u] \rightarrow B_D [y, v])\).
- Axiom of choice: \(\exists f, \exists g, \forall u, \forall v, (A_D (u, guv) \rightarrow B_D [fu, v])\).

**Dynamic behaviour**: agrees to a chain rule.

Mathematical meaning: it’s some kind of approximation.

Outline of the talk

- The Historical Dialectica
- Differentiation and Differentiable Programming.
- Factorizing Dialectica through differential linear logic.
- Dialectica acting on λ-terms.
- Applications and related work.
Differentiable Programming
Differentiation

- Differentiation is finding the best linear approximation to a function at a point.

\[ f \in C^\infty(\mathbb{R}, \mathbb{R}) \]

Chain Rule: \[ D_0(f \circ g) = D_{g(0)}f \circ D_0g \]

- Differentiation is a mathematical operation which needs to be fitted to logical and computer science use.
  - Algorithmic Differentiation: differentiating sequences of many-valued functions efficiently.
  - Differential Linear Logic: Differentiating proofs and \( \lambda \)-terms.
Differentiation

▶ Differentiation is finding the best linear approximation to a function at a point.

$$f \in C^\infty(\mathbb{R}, \mathbb{R})$$

Chain Rule: $$D_0(f \circ g) = D_{g(0)}f \circ D_0g$$

▶ Differentiation is a mathematical operation which needs to be fitted to logical and computer science use.

▶ Algorithmic Differentiation: differentiating sequences of many-valued functions efficiently.
▶ Differential Linear Logic: Differentiating proofs and $\lambda$-terms.
Dialectica verifies the chain rule

Composing the Dialectica interpretation of arrows:

\[(A \Rightarrow B)_D[\phi_1; \psi_1, u_1; v_1] := AD(u_1, \psi_1 u_1 v_1) \Rightarrow BD(\phi_1 u_1, v_1)\]

\[(B \Rightarrow C)_D[\phi_2; \psi_2, u_2; v_2] := BD(u_2, \psi_2 u_2 v_2) \Rightarrow CD(\phi_2 u_2, v_2)\]

\[(A \Rightarrow C)_D[\phi_3; \psi_3, u_3; v_3] := AD(u_3, \psi_3 u_3 v_3) \Rightarrow CD(\phi_3 u_3, v_3)\]

The Dialectica interpretation amounts to the following equations:

\[u_3 = u_1\]
\[v_3 = v_2\]
\[u_2 = \phi_1 u_1\]
\[\psi_3, u_3, v_3 = \psi_1, u_1, v_1\]
\[\phi_2 u_2 = \phi_1, u_1\]
\[v_1 = \psi_2(u_2, v_2)\]

which can be simplified to:

\[\phi_3(u_3) = \phi_2(\phi_1(u_3)) \text{ composition of functions}\]
\[\psi_3(u_3, v_3) = \psi_1(u_3, \psi_2(\phi_1 u_3, v_3)) \text{ composition of their differentials}\]

Thanks to T. Powell for noticing typos here.
But verifying the chain rule does not make you differentiation!

- More modern presentations of Dialectica.

- More Computer Science Friendly presentations of Differentiation.

- Linearity must enter the game.
Curry-Howard for semantics

<table>
<thead>
<tr>
<th>Programs</th>
<th>Logic</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fun (x:A) -&gt; (t:B)</code></td>
<td>Proof of $A \vdash B$</td>
<td>$f : A \rightarrow B$.</td>
</tr>
<tr>
<td>Types</td>
<td>Formulas</td>
<td>Objects</td>
</tr>
<tr>
<td>Execution</td>
<td>Cut-elimination</td>
<td>Equality</td>
</tr>
</tbody>
</table>

**Dialectica**

- Differential $\lambda$-calculus
- Differential Linear Logic
- Differential Categories

*Dialectica is Backward Differentiation in Logic*
And now for something completely different: Automatic Differentiation

How does one compute the differentiation of an algebraic expression, computed as a sequence of elementary operations?

E.g.:

\[ z = y + \cos(x^2) \]

\[ x_1 = x_0^2 \quad x_1' = 2x_0x_0' \]

\[ x_2 = \cos(x_1) \quad x_2' = -x_0' \sin(x_0) \]

\[ z = y + x_2 \quad z' = y' + 2x_2x_2' \]

Derivative of a sequence of instruction

\[ \downarrow \]

sequence of instruction \times sequence of derivatives

**Forward Mode differentiation** [Wengert, 1964]

\( (x_1, x'_1) \rightarrow (x_2, x'_2) \rightarrow (z, z') \).

**Reverse Mode differentiation:** [Speelpenning, Rall, 1980s]

\( x_1 \rightarrow x_2 \rightarrow z \rightarrow z' \rightarrow x_2' \rightarrow x_1' \) while keeping formal the unknown derivative.
Curry-Howard for semantics

The syntax mirrors the semantics.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Logic</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{fun} \ (x:A) \rightarrow (t:B) )</td>
<td>Proof of ( A \vdash B )</td>
<td>( f : A \rightarrow B. )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Types</th>
<th>Formulas</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution</td>
<td>Cut-elimination</td>
<td>Equality</td>
</tr>
</tbody>
</table>

- Programs acts on programs.
  - Functions are higher-order: they act not only on \( \mathbb{R}^n \), but also on \( C^\infty(\mathbb{R}^n, \mathbb{R}) \).

- Programs are typed.
  - Add: \( C^\infty(\mathbb{R}^n, \mathbb{R}) \times C^\infty(\mathbb{R}^n, \mathbb{R}) \rightarrow C^\infty(\mathbb{R}^n, \mathbb{R}) \)

- Everything is interpreted in Categories.
  - Objects are Data
  - Functions are Programs
  - Transformations are functorial:

\[
\mathcal{F}(p_1; p_2) = \mathcal{F}(p_1) ; \mathcal{F}(p_2)
\]

\[
\mathcal{F}(f_2 \circ f_1) = \mathcal{F}(f_2) \circ \mathcal{F}(f_1)
\]
Back to AD: I hate graphs

\[ D_u(f \circ g) = D_{g(u)} f \circ D_u(g) \]

- **Forward Mode differentiation:**
  \[ g(u) \rightarrow D_u g \rightarrow f(g(u)) \rightarrow D_{g(u)} f \rightarrow D_{g(u)} f \circ D_u(g). \]

- **Reverse Mode differentiation:**
  \[ g(u) \rightarrow f(g(u)) \rightarrow D_{g(u)} f \rightarrow D_u(g) \rightarrow D_{g(u)} f \circ D_u(g) \]

The choice of an algorithm is due to complexity considerations:
- **Forward mode** for \( f \circ g : \mathbb{R} \rightarrow \mathbb{R}^n \).
- **Reverse mode** for \( f \circ g : \mathbb{R}^n \rightarrow \mathbb{R} \).

\( \rightsquigarrow \) *Differentiable programming* is a new research area triggered by the advances of deep learning algorithms on neural networks, it tries to attach two very old domains: lambda-calculus and automatic differentiation, with *correctness* and *modularity* goals in mind.
AD from a functorial point of view

\[ D_u(f \circ g) = D_{g(u)}f \circ D_u(g) \]

Non-functorial !!!

How to make differentiation functorial? Make it act on pairs!

\[ f : E \Rightarrow F \]

Forward Mode differentiation:

\[ f : E \Rightarrow E \rightsquigarrow \overrightarrow{D}f : E \Rightarrow E \rightarrow F. \]

\[ \overrightarrow{D}(f) : \begin{cases}  
E \Rightarrow E \rightarrow F \\
    u \mapsto v \mapsto D_u(f)(v)
\end{cases} \]

Functorial forward differentiation:

\[ (f, \overrightarrow{D}(f)) : \begin{cases} 
E \times E \rightarrow F \times F \\
    (a, x) \mapsto (f(a), (D_a f \cdot x))
\end{cases} \]
Reverse AD from a functorial point of view

How to make reverse differentiation functorial?

Make it act on pairs with linear duals!
Reverse functorial differentiation

**Linear Dual**

\[ A^\perp \equiv A \rightarrow \perp \equiv \mathcal{L}(A, \mathbb{R}) \]

- **Reverse Mode differentiation:**

  \[ g(u) \rightarrow f(g(u)) \rightarrow D_{g(u)}f \rightarrow D_{g(u)}f \circ D_u(g) \]

  \[ f : E \Rightarrow F \rightsquigarrow \overset{\leftarrow}{D} f : E \Rightarrow F^\perp \Rightarrow E^\perp. \]

  \[ \overset{\leftarrow}{D}(f) : \left\{ \begin{array}{c} E \Rightarrow F^\perp \rightarrow E^\perp \\
  u \mapsto \ell \mapsto \ell \circ D_u(f) \end{array} \right. \]

  [Mazza, Pagani, POPL2020]

- **Reverse functorial differentiation:**

  \[ (f, \overset{\leftarrow}{D}(f)) : (E \Rightarrow F) \times (E \Rightarrow F^\perp \Rightarrow E^\perp) \]
Reverse functorial differentiation

**Linear Dual**

\[ A^\perp \equiv A \rightarrow \perp \equiv \mathcal{L}(A, \mathbb{R}) \]

▶ **Reverse Mode differentiation:**

\[ g(u) \rightarrow f(g(u)) \rightarrow D_{g(u)}f \rightarrow D_{g(u)}f \circ D_u(g) \]

\[
f : E \Rightarrow F \rightsquigarrow \overset{\leftarrow}{D} f : E \Rightarrow F^\perp \Rightarrow E^\perp.
\]

\[
\overset{\leftarrow}{D}(f) : \begin{cases} 
E \Rightarrow F^\perp & \rightarrow E^\perp \\
u \mapsto \ell \mapsto \ell \circ D_u(f) 
\end{cases}
\]

[Mazza, Pagani, POPL2020]

▶ **Reverse functorial differentiation:**

\[
(f, \overset{\leftarrow}{D}(f)) : (E \Rightarrow F) \times (E \Rightarrow F^\perp \Rightarrow E^\perp)
\]
Reverse functorial differentiation

**Linear Dual**

\[ A^\perp \equiv A \to \perp \equiv \mathcal{L}(A, \mathbb{R}) \]

▸ **Reverse Mode differentiation:**

\[
g(u) \to f(g(u)) \to D_{g(u)} f \to D_{g(u)} f \circ D_u(g)
\]

\[
f : E \Rightarrow F \rightsquigarrow \overset{\leftarrow}{D} f : E \Rightarrow F^\perp \Rightarrow E^\perp.
\]

\[
\overset{\leftarrow}{D}(f) : \begin{cases}
E \Rightarrow F^\perp \to E^\perp \\
u \mapsto \ell \mapsto \ell \circ D_u(f)
\end{cases}
\]

[Mazza, Pagani, POPL2020]

▸ **Reverse functorial differentiation :**

\[
(f, \overset{\leftarrow}{D}(f)) : (E \Rightarrow F') \times (E \Rightarrow F^\perp \Rightarrow E^\perp)
\]
Types!
Programs and variable are **typed**
by logical formulas which describe their behavior

\[ A \rightsquigarrow \exists x : W(A), \forall u : C(A), A_D[x, u] \]

**Witness and counter types** :

\[ C(A \Rightarrow B) = C(A) \times C(B) \]

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

**Reverse Mode differentiation**:

Functorial : \((h, \overleftarrow{D} h) : (A \Rightarrow B) \times (A \Rightarrow B^\perp \rightarrow A^\perp)\)

**However**:

- Having the same type does not mean you’re the same program.
- Some french (linear) logicians have a strong opinion on what proof differentiation should.
Types!

Programs and variable are **typed** by logical formulas which describe their behavior.

\[ A \leadsto \exists x : W(A) , \forall u : C(A) , A_D[x,u] \]

**Witness and counter for implication types:**

\[ C(A \Rightarrow B) = C(A) \times C(B) \]

\[ W(A \Rightarrow B) = (W(A) \Rightarrow W(B)) \times \left( W(A) \Rightarrow C(B) \Rightarrow C(A) \right) \]

**Reverse Mode differentiation:**

Functorial: \((h, \overset{\rightarrow}{D} h) : (A \Rightarrow B) \times (A \Rightarrow B^\perp \rightarrow A^\perp)\)

However:

- Having the same type does not mean you’re the same program.
- Some french (linear) logicians have a strong opinion on what proof differentiation should.
A Linear Logic Refinement
Curry-Howard for semantics

*The syntax mirrors the semantics.*

<table>
<thead>
<tr>
<th>Programs</th>
<th>Logic</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fun (x:A) -&gt; (t:B)</code></td>
<td><code>Proof of A ⊢ B</code></td>
<td><code>f : A → B</code></td>
</tr>
<tr>
<td>Types</td>
<td>Formulas</td>
<td>Objects</td>
</tr>
<tr>
<td>Execution</td>
<td>Cut-elimination</td>
<td>Equality</td>
</tr>
</tbody>
</table>

Doing to proofs everything we do to functions
Curry-Howard for semantics

The syntax mirrors the semantics.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Logic</th>
<th>Semantics</th>
</tr>
</thead>
</table>

Types | Formulas | Objects |
Execution | Cut-elimination | Equality |

Differential λ-calculus [Ehr04]  Differential Linear Logic [Ehrhard06]  Vectorial Models

Linear Logic [Gir87]

λ-calculus  Min. Logic  Normal functors

Doing to proofs everything we do to functions
Linear Logic

Usual Implication

Linear and Non Linear Arrows

\[ A \Rightarrow B = ! A \multimap B \]
\[ C^\infty (A, B) \simeq \mathcal{L}(!A, B) \]

A proof is linear when it uses only once its hypothesis \( A \).

- Notions of ressources which have made their way into programmation through linear types.
- The dynamics of linearity gets encoded through the rules of the ! connective, and its dual ?.

\[ A, B := A \otimes B \mid A \bowtie B \mid A \oplus B \mid A \& B \mid !A \mid ?A \]
Linear Logic

Usual implication

Linear and Non Linear Arrows

\[ A \Rightarrow B = ! A \rightarrow B \]
\[ C^\infty(A, B) \simeq L(!A, B) \]

A proof is linear when it uses only once its hypothesis \( A \).

- Notions of ressources which have made their way into programmation through linear types.
- The dynamics of linearity gets encoded through the rules of the ! connective, and its dual ?.}

\[ A, B := A \otimes B | A \bowtie B | A \oplus B | A \& B | !A | ?A \]
Linear Logic

Usual implication

Linear and Non Linear Arrows

\[ A \Rightarrow B = ! A \multimap B \]
\[ C^\infty(A, B) \simeq \mathcal{L}(!A, B) \]

Linear Implication

Exponential

A proof is linear when it uses only once its hypothesis A.

- Notions of ressources which have made their way into programmation through linear types.
- The dynamics of linearity gets encoded through the rules of the ! connective, and its dual ?.

\[ A, B := A \otimes B | A \bowtie B | A \oplus B | A \& B | !A | ?A \]
Dialectica factorizes through Linear Logic

The call by name arrow

\[ A \Rightarrow B := !A \rightarrow B := ((!A) \otimes B^\perp)^\perp \]

\[
\begin{align*}
\mathbb{W}(A^\perp) &:= \mathbb{C}(A) & \mathbb{C}(A^\perp) &:= \mathbb{W}(A) \\
\mathbb{W}(!A) &:= \mathbb{W}(A) & \mathbb{C}(!A) &:= \mathbb{W}(A) \Rightarrow \mathbb{C}(A) \\
\mathbb{W}(A \otimes B) &:= \mathbb{W}(A) \times \mathbb{W}(B) & \mathbb{C}(A \otimes B) &:= (\mathbb{W}(A) \Rightarrow \mathbb{C}(B)) \times (\mathbb{W}(B) \Rightarrow \mathbb{C}(A))
\end{align*}
\]

Valeria de Paiva, 1989, A dialectica-like model of linear logic.
Differential Linear Logic

\[ \vdash \ell : A \rightarrow B \quad d \]
\[ \vdash \ell : !A \rightarrow B \]

A linear proof

is in particular non-linear.

\[ \vdash f : !A \rightarrow B \quad \bar{d} \]
\[ \vdash D_0 f : A \rightarrow B \]

From a non-linear proof

we can extract a linear proof

\[ f \in C^\infty(\mathbb{R}, \mathbb{R}) \]

Differential interaction nets, Ehrhard and Regnier, TCS (2006)
Exponential rules of Differential Linear Logic

Exponential connectives:

\[ [!A] := C^\infty([A], K)' \quad [?A] := C^\infty([A]', K) \]

\[ \frac{\vdash \Gamma}{\vdash \Gamma, cst_1 : ?A} \quad w \]
\[ \frac{\vdash \Gamma}{\vdash \Gamma, \delta_0 : !A} \quad \bar{w} \]
\[ \frac{\vdash \Gamma, f : ?A, g : ?A}{\vdash \Gamma, f.g : ?A} \quad c \]
\[ \frac{\vdash \Gamma, \ell : ?A}{\vdash \Gamma, \ell : A} \quad d \]
\[ \frac{\vdash \Gamma, \phi : !A}{\vdash \Gamma, \Delta, \psi * \phi : !A} \quad \bar{c} \]
\[ \frac{\vdash \Gamma, x : A}{\vdash \Gamma, D_0(_x)(x) : !A} \quad \bar{d} \]
\[ \frac{?\Gamma \vdash x : A}{?\Gamma \vdash \delta_x : !A} \quad p \]
Differentiation in Differential Linear Logic

The only thing you need to know:

\[
\vdash \Gamma, \delta_u : !A \quad \vdash \Gamma, v : A \quad \vdash \Gamma, D_0(-)(v) : !A
\]

\[
\frac{\vdash \Gamma, \delta_u : !A \quad \vdash \Gamma, v : A}{\vdash \Gamma, \Delta, D_u(-)(v) : !A}
\]

\[
\frac{\vdash \Gamma, v : A \quad \vdash \Gamma, D_0(-)(v) : !A}{\vdash \Gamma, \Delta, D_u(-)(v) : !A}
\]
Dialectica factorizes through Differential Linear Logic

Witnesses are functorial reverse derivative

\[ \mathcal{W}(A \Rightarrow B) = (\mathcal{W}(A) \Rightarrow \mathcal{W}(B)) \times (\mathcal{W}(A) \Rightarrow \mathcal{C}(B) \Rightarrow \mathcal{C}(A)) \]

\[
\begin{align*}
\mathcal{W}(A \otimes B) &:= \mathcal{W}(A) \otimes \mathcal{W}(B) & \mathcal{C}(A \otimes B) &:= (\mathcal{W}(A) \Rightarrow \mathcal{C}(B)) \\
\mathcal{W}(A \multimap B) &:= (\mathcal{W}(A) \multimap \mathcal{W}(B)) & \mathcal{C}(A \multimap B) &:= \mathcal{W}(A \otimes \mathcal{C}(B)) \\
\mathcal{W}(A \& B) &:= \mathcal{W}(A) \& \mathcal{W}(B) & \mathcal{C}(A \& B) &:= \mathcal{C}(A) \oplus \mathcal{C}(B) \\
\mathcal{W}(A \oplus B) &:= \mathcal{W}(A) \oplus \mathcal{W}(B) & \mathcal{C}(A \oplus B) &:= \mathcal{C}(A) \& \mathcal{C}(B) \\
\mathcal{W}(!A) &:= !\mathcal{W}(A) & \mathcal{C}(!A) &:= !\mathcal{W}(A) \Rightarrow \mathcal{C}(A)
\end{align*}
\]

If \( \Gamma \vdash A \) in LL, then \( \mathcal{W}(\Gamma) \vdash \mathcal{W}(A) \) in classical DiLL.

\[
\begin{align*}
\Gamma \vdash A, A &\quad \text{ax} \\
\Gamma \vdash A, !A &\quad \text{\textdagger} \\
\Gamma \vdash ?A, !A &\quad \text{ax} \\
\Gamma \vdash ?A, A, !A &\quad \text{\textdagger} \\
\Gamma \vdash ?A, A &\quad \text{cut}
\end{align*}
\]
Dialectica factorizes through Differential Linear Logic

The economical translation

\[
\begin{align*}
\lbrack A \Rightarrow B\rbrack_e & := !A \multimap B \\
\lbrack A \times B\rbrack_e & := A \& B \\
\lbrack A + B\rbrack_e & := A \oplus B
\end{align*}
\]

ILL \quad \xrightarrow{\mathcal{W}, \mathcal{C}} \quad IDiLL

IDiLL : Intuitionnistic Differential Linear Logic ? Oh no ...
Let's say $x, u, f, g$ are $\lambda$-terms.

The computational Dialectica: a reverse Differential $\lambda$-calculus

"Behind every successful proof there is a program", Gödel's wife
A computational Dialectica

Making Dialectica act on $\lambda$-terms instead of formulas.

$\lambda$-terms with an extra type allowing for sums

\[
\begin{align*}
\Gamma \vdash \emptyset : \mathcal{M}A & \quad \Gamma \vdash m_1 : \mathcal{M}A \quad \Gamma \vdash m_2 : \mathcal{M}A \\
\Gamma \vdash t : A & \quad \Gamma \vdash m_1 \otimes m_2 : \mathcal{M}A \\
\Gamma \vdash \{t\} : \mathcal{M}A & \quad \Gamma \vdash m : \mathcal{M}A \quad \Gamma \vdash f : A \Rightarrow \mathcal{M}B \\
& \quad \Gamma \vdash m >>> f : \mathcal{M}B
\end{align*}
\]

\[
\begin{align*}
\mathcal{W}(A \Rightarrow B) & := (\mathcal{W}(A) \Rightarrow \mathcal{W}(B)) \\
& \quad \times (\mathcal{C}(B) \Rightarrow \mathcal{W}(A) \Rightarrow \mathcal{M}\mathcal{C}(A)) \\
\mathcal{C}(A \Rightarrow B) & := \mathcal{W}(A) \times \mathcal{C}(B)
\end{align*}
\]
Pédrot’s Dialectica Transformation

**Soundness [Ped14]**

If $\Gamma \vdash t : A$ in the source then we have in the target

- $W(\Gamma) \vdash t^\bullet : W(A)$

- $W(\Gamma) \vdash t_x : C(A) \Rightarrow M C(X)$ provided $x : X \in \Gamma$.

**A global and a local transformation**

\[
\begin{align*}
x^\bullet & := x \\
x_x & := \lambda \pi. \{\pi\} \\
x_y & := \lambda \pi. \emptyset \text{ if } x \neq y \\
(\lambda x. t)^\bullet & := (\lambda x. t^\bullet, \lambda \pi x. t_x \pi) \\
(\lambda x. t)_y & := \lambda \pi. (\lambda x. t_y) \pi.1 \pi.2 \\
(t u)^\bullet & := (t^\bullet.1) u^\bullet \\
(t u)_y & := \lambda \pi. (t_y (u^\bullet, \pi)) \otimes ((t^\bullet.2) \pi u^\bullet \gg u_y)
\end{align*}
\]
Flashback: Differential $\lambda$-calculus [Ehrhard, Regnier 04]

Inspired by denotational models of Linear Logic in vector spaces of sequences, it introduces a differentiation of $\lambda$-terms.

$D(\lambda x.t)$ is the **linearization** of $\lambda x.t$, it substitute $x$ linearly, and then it remains a term $t'$ where $x$ is free.

Syntax:

$$\begin{align*}
\Lambda^d &: S, T, U, V ::= 0 \mid s \mid s + T \\
\Lambda^s &: s, t, u, v ::= x \mid \lambda x.s \mid sT \mid Ds.t
\end{align*}$$

Operational Semantics:

$$(\lambda x.s)T \rightarrow_\beta s[T/x]$$

$$D(\lambda x.s) \cdot t \rightarrow_\beta D \lambda x. \frac{\partial s}{\partial x} \cdot t$$

where $\frac{\partial s}{\partial x} \cdot t$ is the **linear substitution** of $x$ by $t$ in $s$. 
Linearity in Linear Logic

**Linearity is about resources:** A proof/program is *linear* iff it uses only once its hypotheses/argument.

<table>
<thead>
<tr>
<th>Linear</th>
<th>Non-linear</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A \vdash A \lor B )</td>
<td>( A \vdash A \land A )</td>
</tr>
<tr>
<td>( \lambda f \lambda x. fxx )</td>
<td>( \lambda x. \lambda f. fxx )</td>
</tr>
</tbody>
</table>

Differentiation is about making a \( \lambda \)-term linear:

\( \rightsquigarrow \) about making a \( \lambda \)-term have a linear usage of its arguments.

\[
\lambda x \lambda f. fxx \rightsquigarrow ?
\]
Linearity in Linear Logic

**Linearity is about resources:** A proof/program is *linear* iff it uses only once its hypotheses/argument.

<table>
<thead>
<tr>
<th>Linear</th>
<th>Non-linear</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A \vdash A \lor B )</td>
<td>( A \vdash A \land A )</td>
</tr>
<tr>
<td>( \lambda f \lambda x. f xx )</td>
<td>( \lambda x. \lambda f. f xx )</td>
</tr>
</tbody>
</table>

Differentiation is about making a \( \lambda \)-term linear:

\[ \rightsquigarrow \text{about making a } \lambda \text{-term have a linear usage of its arguments.} \]

\[ D(\lambda x \lambda f. f xx) \cdot v := \lambda x. \lambda f. vx + ? \]
Linearity in Linear Logic

**Linearity is about resources:** A proof/program is *linear* iff it uses only once its hypotheses/argument.

<table>
<thead>
<tr>
<th>Linear</th>
<th>Non-linear</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \vdash A \lor B$</td>
<td>$A \vdash A \land A$</td>
</tr>
<tr>
<td>$\lambda f \lambda x. fxx$</td>
<td>$\lambda x. \lambda f. fxx$</td>
</tr>
</tbody>
</table>

Differentiation is about making a $\lambda$-term linear:

$\rightsquigarrow$ about making a $\lambda$-term have a linear usage of its arguments.

$$D(\lambda x \lambda f. fxx) \cdot v := \lambda x. \lambda f. vx + \lambda x. \lambda f. Dxv$$
The linear substitution ... 

... which is not exactly a substitution

\[
\frac{\partial y}{\partial x} \cdot t = \begin{cases} 
  t & \text{if } x = y \\
  0 & \text{otherwise}
\end{cases}
\]

\[
\frac{\partial}{\partial x} (tu) \cdot s = (\frac{\partial t}{\partial x} \cdot s)u + (Dt \cdot (\frac{\partial u}{\partial x} \cdot s))u
\]

\[
\frac{\partial}{\partial x} (\lambda y.s) \cdot t = \lambda y \cdot \frac{\partial s}{\partial x} \cdot t
\]

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

\[
\frac{\partial 0}{\partial x} \cdot t = 0
\]

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

\[
\frac{\partial s}{\partial x} \cdot t \text{ represents } s \text{ where } x \text{ is linearly (i.e. one time) substituted by } t.
\]
The linear substitution ...

The computational Dialectica

\[ \frac{\partial y}{\partial x} \cdot t = \begin{cases} t & \text{if } x = y \\ 0 & \text{otherwise} \end{cases} \]

\[ \frac{\partial}{\partial x} (tu) \cdot s = (\frac{\partial t}{\partial x} \cdot s)u + (Dt \cdot (\frac{\partial u}{\partial x} \cdot s))u \]

\[ x_y \cdot \pi = \begin{cases} \pi & \text{if } x = y \\ \emptyset & \text{otherwise} \end{cases} \]

\[ (t \ u)_y := \lambda \pi. (t_y (u^\ast, \pi)) \oslash ((t^\ast.2) \pi u^\ast \gg u_y) \]

\[ \frac{\partial}{\partial x} (\lambda y.s) \cdot t = \lambda y. \frac{\partial s}{\partial x} \cdot t \]

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

\[ \frac{\partial 0}{\partial x} \cdot t = 0 \]

\[ \frac{\partial}{\partial x} (s + u) \cdot t = \frac{\partial s}{\partial x} \cdot t + \frac{\partial u}{\partial x} \cdot t \]
Tracking differentiation in Dialectica

\[ x_x := \lambda \pi. \{ \pi \} \quad x^\bullet := x \]

\[ x_y := \lambda \pi. \varnothing \quad \text{if } x \neq y \]

\[ (\lambda x. t)_y := \lambda \pi. (\lambda x. t_y) \pi.1 \pi.2 \]

\[ (t u)_y := \lambda \pi. (t_y (u^\bullet, \pi)) \oplus ((t^\bullet.2) u^\bullet \pi \ggg u_y) \]
Tracking differentiation in Dialectica

\[
\begin{align*}
    x_x & := \lambda\pi. \{\pi\} & x^\bullet & := x \\
    x_y & := \lambda\pi. \emptyset \text{ if } x \neq y & (\lambda x. t)^\bullet & := (\lambda x. t^\bullet, \lambda x\pi. t_x \pi) \\
    (\lambda x. t)_y & := \lambda\pi. (\lambda x. t_y) \pi.1 \pi.2 & (t u)^\bullet & := (t^\bullet.1) u^\bullet \\
    (t u)_y & := \lambda\pi. (t_y (u^\bullet, \pi)) \otimes (t^\bullet.2) u^\bullet \pi \gg= u_y
\end{align*}
\]
Tracking differentiation in Dialectica

\[
x_x := \lambda \pi. \frac{\partial x}{\partial x} \cdot \pi \\
x_y := \lambda \pi. \frac{\partial x}{\partial y} \cdot \pi \quad \text{if } x \neq y \\
(\lambda x. t)_y := \lambda \pi. (\lambda x. t_y) \pi.1 \pi.2 \\
(t u)_y := \equiv (\lambda x.(t x)_\cdot) u_\cdot
\]

That’s reverse differentiation

- \((\_)_\cdot.2\) obeys the chain rule, \((\_)_\cdot\) is the functorial differentiation.
- \(t_x\) is contravariant in \(x\), representing a reverse linear substitution.

Theorem [K. Pédrot 22]

\[
[u \gg > t_x[\Gamma \leftarrow r_\cdot]] \equiv_{\beta,\eta} \lambda z. ([u] ((\partial x.t[\Gamma \leftarrow r])z))
\]
Tracking differentiation in Dialectica

\[
\begin{align*}
xx & := \lambda \pi. \frac{\partial x}{\partial x} \cdot \pi \\
x^* & := x \\
xy & := \lambda \pi. \frac{\partial x}{\partial y} \cdot \pi \quad \text{if} \ x \neq y \\
(\lambda x. t)^* & := (\lambda x. t^*, \lambda x\pi. t_x \pi) \\
(\lambda x. t)_y & := \lambda \pi. (\lambda x. t_y) \pi.1 \pi.2 \\
(tu)^* & \equiv (\lambda x. (tx)^*)u^*
\end{align*}
\]

That’s reverse differentiation

- \((\_)^*2\) obeys the chain rule, \((\_)^*\) is the functorial differentiation.
- \(t_x\) is contravariant in \(x\), representing a reverse linear substitution.

Theorem [K. Pédrot 22]

\[
[u \ggg t_x[\Gamma \leftarrow \vec{r}^*]] \equiv_{\beta, \eta} \lambda z. ([u] ((\partial x.t[\Gamma \leftarrow \vec{r}])z))
\]
Dialectica is differentiation in categories

That’s already known through lenses!
What’s categorical differentiation?

To cook a good differential category, one needs:

- A category of regular/continuous/non-linear functions
  
  \[ C(A, B) = !A \to B \]

- A category of linear functions, in which differentiation embeds
  
  \[ \mathcal{L}(A, B) = A \to B. \]

- Something which linearizes:
  
  \[ \bar{d} : A \to !A \]

- A notion of duality, if one wants to encode reverse differentiation.
  
  \[ \leadsto \text{Basically, one wants a categorical model of DiLL.} \]
Dialectica categories

Categories representing specific relations

Consider a category $\mathcal{C}$. $\text{Dial}(\mathcal{C})$ is constructed as follows:

- **Objects**: relations $\alpha \subseteq U \times X$, $\beta \subseteq V \times Y$.

- **Maps from $\alpha$ to $\beta$**:

$$(f : U \to V, F : U \times Y \to X)$$

- **Composition**: the chain rule!

Consider

$$(f, F) : \alpha \subseteq (A, X) \to \beta \subseteq (B, Y)$$

and

$$(g, G) : \beta \subseteq (B, Y) \to \gamma \subseteq (C, Z)$$

two arrows of the Dialectica category. Then their composition is defined as

$$(g, G) \circ (f, F) := (g \circ f, (a, z) \mapsto F(a, G(f(a), z))).$$
Dialectica categories through Differential Categories

In a \(*\)-autonomous differential category:

\[ \partial : \text{Id} \otimes ! \to ! \]

\[ \mathcal{L}(B \otimes A, C^\perp) \simeq \mathcal{L}(A, (B \otimes C)^\perp) \]

from \( f : !A \to B \) one constructs:

\[ \overleftarrow{D}(f) \in \mathcal{L}(!A \otimes B^\perp, A^\perp). \]

Dialectica categories factorize through differential categories

If \( \mathcal{L} \) is a model of DiLL such that \( \mathcal{L}! \) has finite limits:

\[
\begin{align*}
\mathcal{L}! & \to \mathcal{D}(\mathcal{L}!) \\
A & \mapsto A \times A^\perp \\
f & \mapsto (f, \overleftarrow{D}(f))
\end{align*}
\]

We have an obvious forgetful functor:

\[
\mathcal{U} : \left\{ \begin{array}{c}
\mathcal{D}(\mathcal{L}!) \to \mathcal{L}! \\
\alpha \subseteq A \times X \mapsto A \\
(f, F) \mapsto f
\end{array} \right. 
\]
Recap

<table>
<thead>
<tr>
<th>Programs</th>
<th>Logic</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{fun } (x:A) \rightarrow (t:B) )</td>
<td>Proof of ( A \vdash B )</td>
<td>( f : A \rightarrow B ).</td>
</tr>
</tbody>
</table>

- **Types**
- **Formulas**
- **Objects**
- **Execution**
- **Cut-elimination**
- **Equality**

- **\( \lambda \)-calculus** [Ehr04]
- **Linear Logic** [Gir87]
- **Differential Linear Logic** [Ehrhard06]
- **Vectorial Models**
- **\( \lambda \)-calculus**
- **Min. Logic**
- **Normal functors**
Recap

<table>
<thead>
<tr>
<th>Programs</th>
<th>Logic</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fun (x:A) -&gt; (t:B)</code></td>
<td><code>Proof of A ⊢ B</code></td>
<td><code>f : A → B</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Types</th>
<th>Formulas</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution</td>
<td>Cut-elimination</td>
<td>Equality</td>
</tr>
</tbody>
</table>

- Differential \(\lambda\)-calculus [Ehr04]
- Differential Linear Logic [Ehrhard06]
- Vectorial Models
- Linear Logic [Gir87]
- \(\lambda\)-calculus
- Min. Logic
- Normal functors
Recap

<table>
<thead>
<tr>
<th>Programs</th>
<th>Logic</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{fun} (x:A) \rightarrow (t:B))</td>
<td>Proof of (A \vdash B)</td>
<td>(f : A \rightarrow B).</td>
</tr>
</tbody>
</table>

Types  
Formulas  
Execution  
Cut-elimination  
Objects  
Equality

Differential \(\lambda\)-calculus [Ehr04]  
Differential Linear Logic [Ehrhard06]  
Linear Logic [Gir87]  
Vectorial Models

Automatic Differentiation [80s]  
Dialectica [Göd58]  
\(\lambda\)-calculus  
Min. Logic  
Normal functors
Recap

<table>
<thead>
<tr>
<th>Programs</th>
<th>Logic</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{fun } (x:A) \rightarrow (t:B) )</td>
<td>Proof of ( A \vdash B )</td>
<td>( f : A \rightarrow B ).</td>
</tr>
<tr>
<td>Types</td>
<td>Formulas</td>
<td>Objects</td>
</tr>
<tr>
<td>Execution</td>
<td>Cut-elimination</td>
<td>Equality</td>
</tr>
</tbody>
</table>

- Differentiable Programming
- Differential λ-calculus [Ehr04]
- Automatic Differentiation [80s]
- λ-calculus
- Min. Logic
- Normal functors
- Differential Linear Logic [Ehrhard06]
- Vectorial Models
- Linear Logic [Gir87]
- Dialectica [Göd58]
Recap

<table>
<thead>
<tr>
<th>Programs</th>
<th>Logic</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{fun} \ (x:A) \to (t:B) )</td>
<td>( \text{Proof of } A \vdash B )</td>
<td>( f : A \to B ).</td>
</tr>
<tr>
<td>Types</td>
<td>Formulas</td>
<td>Objects</td>
</tr>
<tr>
<td>Execution</td>
<td>Cut-elimination</td>
<td>Equality</td>
</tr>
</tbody>
</table>

- Linear Logic \([\text{Gir87}]\)
- Vectorial Models
- Linear Logic \([\text{Gir87}]\)
- Differential Linear Logic \([\text{Ehrhard06}]\)
- Differentiable Linear \(\lambda\)-calculus \([\text{Ehr04}]\)
- Dialectica \([\text{Göd58}]\)
- Normal functors
- Min. Logic
- \(\lambda\)-calculus
- Automatic Differentiation \([80s]\)
- Differentiable Programming

A good point for logicians: Gödel invented Dialectica 40 years before reverse differentiation was put to light
Conclusion and applications
Take home message:

**Dialectica is functorial reverse differentiation**, extracting intensional local content from proofs.

A new semantical correspondence between computations and mathematics: **intentional meaning** of program is **local behaviour** of functions.

<table>
<thead>
<tr>
<th>Program</th>
<th>Proof</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantitative Resources</td>
<td>Classical Principles</td>
<td>Linearity Differentiation</td>
</tr>
<tr>
<td>Control</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Related work and potential applications:

- **Markov’s principle** and delimited continuations on positive formulas.
- **Proof mining** and backpropagation.
- **Bar Induction** and Taylor Exponentiation.
Dialectica is differentiation ...

... We knew it already!

The codereliction of differential proof nets: In terms of polarity in linear logic [23], the ∀→-free constraint characterizes the formulas of intuitionistic logic that can be built only from positive connectives (⊕, ⊗, 0, 1, !) and the why-not connective (“?”). In this framework, Markov’s principle expresses that from such a ∀→-free formula A (e.g., ? ⊕ x (?A(x) ⊗ ?B(x))) where the presence of “?” indicates that the proof possibly used weakening (efq or throw) or contraction (catch), a linear proof of A purged from the occurrences of its “?” connective can be extracted (meaning for the example above a proof of ⊕ x (A(x) ⊗ B(x))). Interestingly, the removal of the “?” , i.e. the steps from ?P to P, correspond to applying the codereliction rule of differential proof nets [24].

**Differentiation**: (?P = (P → ⊥) ⇒ ⊥) → ((P → ⊥) → ⊥) ≡ P

---

Hugo Herbelin, “An intuitionistic logic that proves Markov’s principle”, LICS ’10.
Markov’s principle is proved by allowing catch and throw operations on hereditary positive formulas.

**Figure 3. Proof of MP**
Effective moduli from ineffective uniqueness proofs. An unwinding of de La Vallée Poussin’s proof for Chebycheff approximation*

Ulrich Kohlenbach
Fachbereich Mathematik, J.W. Goethe Universität
Robert-Mayer Str. 6 10, 6000 Frankfurt am Main, FRG

Abstract
We consider uniqueness theorems in classical analysis having the form

\[(+) \forall u \in U, v_1, v_2 \in V_u \left( G(u, v_1) = 0 = G(u, v_2) \rightarrow v_1 = v_2 \right) ,\]

where \( U, V \) are complete separable metric spaces, \( V_u \) is compact in \( V \) and \( G : U \times V \rightarrow \mathbb{R} \) is a constructive function.

If \((+)\) is proved by arithmetical means from analytical assumptions

\[ (+++) \forall x \in X \exists y \in Y_x \forall z \in Z \left( F(x, y, z) = 0 \right) \]

only (where \( X, Y, Z \) are complete separable metric spaces, \( Y_x \subset Y \) is compact and \( F : X \times Y \times Z \rightarrow \mathbb{R} \) constructive), then we can extract from the proof of \((+++) \rightarrow (+)\) an effective modulus of uniqueness, i.e.

\[ (+++) \forall u \in U, v_1, v_2 \in V_u, k \in \mathbb{N} \left( |G(u, v_1)|, |G(u, v_2)| \leq 2^{-\Phi u_k} \rightarrow d_V(v_1, v_2) \leq 2^{-k} \right) .\]
Proof Mining

Markov’s principle and the independence of premises are necessary for most of mathematical analysis proofs:

Proof mining allows to refine these proofs by taking away these principles as guaranteed by (some variant of) Dialectica’s transformation.

**Conjecture**

Does it differentiate the function \((\epsilon \rightarrow \eta)\) in:

\[
\forall u, v_1 v_2, \forall \epsilon > 0, \exists \eta > 0, |G(u, v_1) - G(u, v_2)| < \eta \Rightarrow d_V(v_1, v_2) < \epsilon
\]

Is proof mining (based on) reverse differentiation applied to proofs?

What else can we explain by differentiation?
Thank you for Listening !