Towards a Non-Commutative Logic of Effects

Noam Zeilberger

Work-in-Progress

with

Jonas Frey & Paul-André Melliès

England, December 2010
What's up with Linear Logic?

After years of promise, why is linear logic still not a common language for talking about computation?

The leading suspects:

1) Involution, negation \( (A^{\perp\perp} = A) \) ?
2) Exchange \( (A \oplus B = B \oplus A) \) ?

I suggest (1) is harmless ultimately, but (2) is deadly:

effects do not (in general) commute!
Okay, so let's use non-commutative linear logic — problem solved?

We'd like to do better: the tools may already be there for understanding why this is a nice conceptual space, and how it relates to other nice spaces.

Main idea of this talk: it is possible to derive a non-commutative monoidal structure from an abstract duality between types & contexts.

Caveat: this is work in progress, X% baked for some X probably significantly below 50.
A Beautiful Idea

A type is a set of values

\[ B = \{ \text{true}, \text{false} \} \quad \text{IN} = \{ 0, 1, 2, \ldots \} \]

\[ \text{string} = \{ \text{"foo"}, \text{"bar"}, \text{"baz"} \ldots \} \]

A map \( A \rightarrow B \) transforms \( A \)-values to \( B \)-values

and : \( B \times B \rightarrow B \)
and \( (\text{true}, \text{true}) = \text{true} \)
and \( (\text{true}, \text{false}) = \text{false} \)
and \( (\text{false}, \text{true}) = \text{false} \)
and \( (\text{false}, \text{false}) = \text{false} \)

Length : string \( \rightarrow \) \( \text{IN} \)
length "foo" \( \equiv \) 3
length "bar" \( \equiv \) 3
length "baz" \( \equiv \) 3

...
A Slightly Less Naive Idea

A type is a presheaf of values

i.e., a family of sets of values \( \forall v : P^\tau \), indexed by context, compatible with substitution:

\[
\frac{\Gamma' \vdash \sigma : \Gamma \quad \Gamma \vdash v : P}{\Gamma' \vdash \sigma \circ v : P}
\]

i.e., a contravariant functor \( P : C^{op} \to \text{Set} \) over some "category of contexts" \( C \).

Collectively, these types can be organized into the functor category \( [C^{op}, \text{Set}] \), which we call \( \hat{C} \).
Another Beautiful Idea

A type is a set of continuations

Intuition: an object is defined by the observations you can make on it.

(e.g., functions are defined by their behaviour on application, pairs are defined by their projections)

A map $A \to B$ transforms $B$-continuations to $A$-continuations

(some concrete Haskell code in a few slides)
And now less naively...

A type is a covariant presheaf of continuations

i.e., a family of sets of continuations $\forall k: N^k \Delta$, indexed by "co-contexts" ("answer type"), compatible with postcomposition:

$$
\dfrac{N^k \Delta \quad \Delta \circ \Delta'}{N^k \Delta'}
$$

i.e., a covariant functor $N: C \longrightarrow \text{Set}$

over some "category of co-contexts" $C$.

Collectively, these types can be organized into the functor category $[C, \text{Set}]^{op}$, which we call $C$.
An analogy

Polarity (in linear logic)
is like
“Isbell Duality”
Isbell Conjugacy

\[ \mathcal{C}^\circ \neg \triangleright \mathcal{C} \]

\[ \mathcal{C}^\circ \neg \triangleright \mathcal{C} \]

\[ \mathcal{C} = [\mathcal{C}, \mathcal{V}]^{\circ} \]

\[ \text{where, } \gamma B = \mathcal{C}([-), B) \]
\[ \gamma A = \mathcal{C}(A, -) \]
An Equation

\[ \text{intvar} \stackrel{\text{def}}{=} \text{intexp} \& \text{intacc} \]

John Reynolds, "Replacing Complexity with Generality: The Programming Language Forsythe"
An Equation

applying the Girard-Reynolds isomorphism...

\[ \text{intvar} \overset{\text{def}}{=} ! (\text{intexp} \& \text{intacc}) \]

John Reynolds, "Replacing Complexity with Generality: The Programming Language Forsythe"

where \[ \text{intexp} \overset{\text{def}}{=} \uparrow \text{int} \]
\[ \text{intacc} \overset{\text{def}}{=} \text{int} \rightarrow \uparrow 1 \]
Negative Types in Haskell

data (\&) n, n_2 x where
  P_1 :: n_1 x \rightarrow (n_1 \& n_2) x
  P_2 :: n_2 x \rightarrow (n_1 \& n_2) x

data (\rightarrow) p, n_2 x where
  App :: p, n_2 x \rightarrow (p \rightarrow n_2) x

data \uparrow p x where
  Plug :: (p \rightarrow x) \rightarrow \uparrow p x

newtype Value n = Val \& runVal ::
  forall r. n \rightarrow r x

plug :: n \star \rightarrow Value n \rightarrow x

k 'plug' v = runVal v k
Programming w/ Variables

data (!) n x where
    Return :: x → ! n x
    Unfold :: n (! n x) → ! n x

"Free monad" construction

type Variable s = ! (↑s & l(s → ↑l))

new :: s → Value (Variable s)
new s = Val new'
    where
        new' (Unfold (P₁ (Plug ks))) = ks s 'plug' new s
        new' (Unfold (P₁ (App s' (Plug k)))) = k ()
            'plug' new s'

        new' (Return x) = x

eexample = Unfold $ P₁ $ Plug $ \s →
            Unfold $ P₂ $ App (s+2) $ Plug $ l() →
            Unfold $ P₁ $ Plug $ \s' →
                Return (λs → s')

fifteen = eexample 'plug' new 3

(c.f. sigfpe's "Programming with impossible functions")

Exercise: write reify :: Variable s x → (l(s → (x,s)))
and reflect :: (l(s → (x,s))) → Variable s x
Types - as - Bimodules
(“Profunctors”, “Distributors”)

Let $\mathcal{L}$ and $\mathcal{R}$ be categories related by a bimodule

$\#: \mathcal{L}^{op} \times \mathcal{R} \rightarrow \text{Set}$

(for more generally “$V$”)

A positive type $P$ is an $(\mathcal{L}, \mathcal{R})$-bimodule which “represents” a “set of values”, i.e.

$P \in [\mathcal{L}^{op} \times \mathcal{R}, \text{Set}] = [\mathcal{R} \times \mathcal{L}^{op}, \text{Set}]$

= $[\mathcal{R}, [\mathcal{L}^{op}, \text{Set}]]$

= $[\mathcal{R}, \mathcal{L}]$

A negative type $N$ is an $(\mathcal{L}, \mathcal{R})$-bimodule which “represents” a “set of continuations”, i.e.

$N \in [\mathcal{L}^{op} \times \mathcal{R}, \text{Set}]^{op} = [\mathcal{L}^{op}, [\mathcal{R}, \text{Set}]]^{op}$

= $[\mathcal{L}, [\mathcal{R}, \text{Set}]^{op}]$

= $[\mathcal{L}, \mathcal{R}]$
Sequent Calculus
Intuition

\[ \#(\Gamma, \Delta) = \text{"} \Gamma \vdash \Delta \text{"} \]
\[ P(\Gamma, \Delta) = \text{"} \Gamma \vdash [P] \Delta \text{"} \]
\[ N(\Gamma, \Delta) = \text{"} \Gamma[N] \vdash \Delta \text{"} \]

Other examples...

- \( L = R = C \), \( \# = C(-, -) \)
- \( L = C, R = 1 \), \( \#(A, *) = C(A, \bot) \)
- \( L = [C, C]^\text{op}, R = C \), \( \#(F, X) = F(X) \)
Bimodule Semantics of Polarised Linear Logic

assume \( \mathcal{L} \) and \( \mathcal{R} \) are monoidal categories
(overloading operations \( \cdot \) and \( \varepsilon \))

\[
P_1 \otimes P_2 (\Gamma, \Delta) = \exists \Gamma', \Delta, \Delta'. P_1 (\Gamma, \Delta) \times P_2 (\Gamma', \Delta')
\]

\[
1 (\Gamma, \Delta) = \mathcal{L} (\Gamma, \varepsilon) \times \mathcal{R} (\delta, \Delta)
\]

\[
P_1 \cdot P_2 (\Gamma, \Delta) = P_1 (\Gamma, \Delta) \cdot P_2 (\Gamma, \Delta)
\]

\[
O (\Gamma, \Delta) = \emptyset
\]

\[
N_1 \land N_2 (\Gamma, \Delta) = \exists \Gamma, \Delta, \Delta'. N_1 (\Gamma, \Delta) \times N_2 (\Gamma', \Delta')
\]

\[
\bot (\Gamma, \Delta) = \mathcal{L} (\Gamma, \varepsilon) \times \mathcal{R} (\delta, \Delta)
\]

\[
N_1 \lor N_2 (\Gamma, \Delta) = N_1 (\Gamma, \Delta) \lor N_2 (\Gamma, \Delta)
\]

\[
T (\Gamma, \Delta) = \emptyset
\]

\[
N^+ (\Gamma, \Delta) = N (\Gamma, \Delta)
\]

\[
P^+ (\Gamma, \Delta) = P (\Gamma, \Delta)
\]

\[
\downarrow N (\Gamma, \Delta) = \forall \Gamma', \Delta'. N (\Gamma', \Delta') \rightarrow (\Gamma \cdot \Gamma', \Delta' \cdot \Delta)
\]

\[
\uparrow P (\Gamma, \Delta) = \forall \Gamma', \Delta'. P (\Gamma', \Delta') \rightarrow (\Gamma \cdot \Gamma', \Delta' \cdot \Delta)
\]
Isbell Revisited

\[ [R, \hat{L}] = \text{Pos} \]

\[ \text{Neg} = [\hat{L}, \hat{R}] \]

\[ L \times R^\circ \]

where \( \#^+ = \#(\Gamma \dashv, \cdots, \Delta) = \#^- \)
Building a more abstract Picture

Proof theory is about the interaction of types and contexts.

In particular, types can be placed inside of contexts.

Where does the (monoidal) structure of contexts come from?
Suppose that a positive type 
\[ \mathcal{A} \xrightarrow{P} \hat{\mathcal{L}} \]
can be lifted along \( \#^* \)
\[ \mathcal{L} \]
\[ \#^* \]
\[ \mathcal{A} \]
to yield an operation \( P^* : \mathcal{A} \rightarrow \mathcal{A} \) together with a 2-cell \( \eta : P \Rightarrow \#^* P^* : \)
\[ \mathcal{A} \xrightarrow{P} \hat{\mathcal{L}} \]
\[ \#^* \]
\[ \mathcal{A} \]
(Left Kan lift)

We can view \( P^* \) as the operation of extending the right-context with \( P \), and \( \eta \) precisely as the "focalisation rule":
\[
\frac{\Gamma \vdash P \Delta}{\Gamma \vdash P, \Delta} \eta
\]
Dually...

\[ \begin{array}{ccc}
L & \overset{N}{\longrightarrow} & R \\
\downarrow N^* & \nearrow \varepsilon & \downarrow \# \\
\varepsilon & & (\text{right Kan lift})
\end{array} \]

\[
\frac{\Gamma[N] \vdash \Delta}{\Gamma, N \vdash \Delta} \varepsilon
\]
Type-Context Adjunction

More generally, for any \( L: L \rightarrow L \), define
\( p^L \) as a left adjoint "at \( p \)" of postcomposition with \( \#(L, -) \), i.e. such that
\[
\text{End}(\mathcal{R})(p^L, R) \cong [\mathcal{R}, \mathcal{L}](p, \#(L, R))
\]

Likewise, for any \( R: R \rightarrow R \), define
\( N^R \) as a right adjoint "at \( N \)" of precomposition with \( \#(-, R) \), i.e. such that
\[
\text{End}(\mathcal{L})(L, N^R) \cong [\mathcal{L}, \mathcal{R}](\#(L, R), N)
\]

\( p^L \) is a sort of "weighted colimit"
\( N^R \) is a sort of "weighted limit"
Monoidal Structure Revisited

A monoidal product on \( \mathcal{L} \) and \( \mathcal{R} \) can be derived for contexts composed of types:

\[ \varepsilon \cdot \Delta = \Delta \]

\[ (p^k \Delta_1) \cdot \Delta_2 = p^k (\Delta_1 \cdot \Delta_2) \]

\[ \Gamma \cdot \varepsilon = \Gamma \]

\[ \Gamma_1 \cdot (N_R \Gamma_2) = N_R (\Gamma_1 \cdot \Gamma_2) \]
Structural Properties

In general, we won't have exchange:

\[
\frac{\Gamma, N_l, N_r + \Delta}{\Gamma, N_l, N_r - \Delta} \quad \frac{\Gamma + P_1, P_2, \Delta}{\Gamma + P_2, P_1, \Delta}
\]

(only when \( P_1 \) and \( P_2 \) (\( N_l \) and \( N_r \)) commute)

In general, we won't have weakening contraction:

\[
\frac{\Gamma + \Delta}{\Gamma, \Delta} \quad \frac{\Gamma, N_l, N_r + \Delta}{\Gamma, N_l, N_r - \Delta} \quad \frac{\Gamma + P, P, \Delta}{\Gamma + P, P, \Delta} \quad \frac{\Gamma + \Delta}{\Gamma + P, \Delta}
\]

(only when \( P \) (\( N \)) defines a (co)monad on \( \mathcal{L} \))
General Principle:
Definition - by - Kan extension

\[ L_0^{op} \times R_0 \]

\[ i \quad \text{k} \quad p_0 \]

\[ L_0^{op} \times R \rightarrow \text{Set} \]

\[ P(\Gamma, \Delta) = \exists \Gamma_0, \Delta_0. \quad p_0(\Gamma_0, \Delta_0) \]

An abstract "subformula property"
The Exponentials

a candidate definition:

\[ !N (\Gamma, \Delta) = \#(\Gamma, \Delta) + N(\Gamma, \Gamma !_N, \Delta) \]
\[ ?P (\Gamma, \Delta) = \#(\Gamma, \Delta) + P(\Gamma, \Gamma !_?P, \Delta) \]

caveats:

- Why is this well-defined?

- These may or may not be related to the exponentials in LL
  (cf. Melliès-Tabareau-Tasson, "An explicit formula for the free exponential modality of linear logic")
A Proto-Calculus

Contents and context transformers

\[ \Gamma ::= X | L(\Gamma) \quad L ::= \alpha | \eta R \]
\[ \Delta ::= Y | R(\Delta) \quad R ::= \beta | \eta P \]

Judgments

\[ \Gamma_1 \vdash \Gamma_2 \quad L_1 \vdash L_2 \quad R_1 \vdash R_2 \quad \Delta_1 \vdash \Delta_2 \]
\[ \Gamma[N] \vdash \Delta \quad \Gamma \vdash \Delta \quad \Gamma \vdash [P] \Delta \]

Rules

\[ X \vdash X \quad \alpha \vdash \alpha \quad \beta \vdash \beta \quad Y \vdash Y \]
\[ \Gamma \vdash [P] \Delta \quad L(\Gamma) \vdash \eta R(\Delta) \]
\[ \Gamma \vdash [P] \Delta \quad \eta R(\Delta) \vdash L(\Gamma) \]
\[ R_1 \vdash R_2 \quad \Delta_1 \vdash \Delta_2 \]
\[ \Gamma[N] \vdash \Delta \quad \eta R(\Delta) \vdash L(\Gamma) \]
\[ \Gamma \vdash [P] \Delta \quad \eta R(\Delta) \vdash L(\Gamma) \]
\[ \Gamma \vdash [P] \Delta \quad \eta R(\Delta) \vdash L(\Gamma) \]
\[ \Gamma \vdash [P] \Delta \quad \eta R(\Delta) \vdash L(\Gamma) \]