1 Introduction

In this blog post, we provide an overview of the *Dialectica categories* and explore their connection to **Bel**, the category of *backward error lenses.*

- Developed by **de1991dialectica**, the Dialectica categories are categorical models of intuitionistic linear logic. Fixing a base category **C**, we can construct a Dialectica category, abbreviated **DC**, such that logical formulas can be interpreted as objects in **DC** and deductions can be interpreted as morphisms.
- The objects of **DC** are (set-theoretic) *relations* on objects from **C**.
- The morphisms in **DC** are pairs of morphisms in **C** satisfying a certain condition. This category is one of the earliest examples of *lenses*, a data accessor structure we see in functional programming.

Our goal is to use the Dialectica categories to inform our construction of a similar category, **Bel**, from **kellison2025bean**.

- **Bel** is the category used to interpret a linear type system, Bean, which synthesizes bounds on roundoff error for numerical programs. We show soundness of these error bounds by interpreting Bean typing judgments as morphisms in **Bel**.
- The objects of **Bel** are metric spaces, which may be rewritten as families of relations on a set.
- The morphisms in **Bel** are triples of functions reminiscent of lenses.

Sound familiar? We thought so. Now, we're going to introduce linear logic and the categories **DC**, and compare them to Bean and the category **Bel**.

2 Linear logic

2.1 Deductive logic

A deductive logic is just a set of *axioms* and *inference rules* which tell you how to write down *deductions.* An axiom looks like this:

$$\frac{}{\Gamma, A \vdash A}$$

To the left of $\vdash$ is the *context*, the set of logical formulas that we are given. Here, Γ is a set of formulas, A is a single formula, and Γ, A combines them. To the right of $\vdash$ is the conclusion, A. The $\vdash$ means "proves." So, this axiom says: "Given Γ and A, we can prove A." This seems pretty self-evident, which is why it's an axiom.

An inference rule looks like this:

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \text{ and } B}$$

Here, Γ and Δ are two contexts, and Γ, Δ is a context combining them. The deductions $\Gamma \vdash A$ and $\Delta \vdash B$ above the bar are requirements for deducing $\Gamma, \Delta \vdash A$ and B. So, this rule says, "If given Γ we can prove A, and given Δ we can prove B, then given both Γ and Δ, we can prove A and B." The idea is that any deduction we're able to prove from these rules should be "valid" (consistency), and any valid deduction should be provable from these rules (completeness).

2.2 Intuitionistic linear logic

Linear logic is a *substructural* logic, which means it's missing at least one of the common *structural* rules found in other logics. In this case, linear logic is missing *weakening* and *contraction.*

$$\frac{\Gamma \vdash B}{\Gamma, \Delta \vdash B}\ (\textsc{Weakening}) \qquad\qquad \frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\ (\textsc{Contraction})$$

Weakening says that if we can prove B from a set of assumptions Γ, then if we are given even more assumptions, we can still prove B. Contraction says that we don't need to use assumptions twice. These rules make sense if we interpret formulas as true statements about the world, e.g. A is "it is raining" and B is "the ground is wet." However, in linear logic, we interpret formulas as *finite resources that must be consumed.* So $\Gamma \vdash A$ means "given exactly all the resources in Γ, we can produce an A." Now, weakening and contraction don't make sense. If $\Gamma \vdash B$, then we cannot conclude $\Gamma, \Delta \vdash B$, because B does not need a Δ to be produced. If $\Gamma, A, A \vdash B$, then B needs two copies of A, and we cannot produce B from just one A. Linear logic is nice because it can model real-world situations, like chemical reactions and concurrent processes.

Intuitionistic logic means that we can't assume the law of the excluded middle (for all A, either A or not A) nor double negation (if not (not A), then A). In a nutshell, this means all our proofs must be constructive. A deeper overview of the history of linear logic and its broader impact on computer science can be seen in **sep-logic-linear**. For now, consider these four inference rules from intuitionistic linear logic:

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\ (\otimes\ \textsc{Intro}) \qquad\qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}\ (\&\ \textsc{Intro})$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\ (\multimap\ \textsc{Intro}) \qquad\qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B}\ (\oplus\ \textsc{Intro-R})$$

The first rule, ($\otimes$ Intro), says that if we have $\Gamma \vdash A$, and $\Delta \vdash B$, then $\Gamma, \Delta \vdash A$ "and" B, where we have enough resources to produce both A and B at the same time. Alternatively, (& Intro) says that if $\Gamma \vdash A$, and $\Gamma \vdash B$, then $\Gamma \vdash A$ "and" B in the sense that Γ can produce either A or B, but not at the same time. This is because we are in linear logic, and A and B each use up all the resources in Γ. Next, $\multimap$ is the linear logic version of implication. So ($\multimap$ Intro) says that if we have $\Gamma, A \vdash B$, then if we have Γ, we just need another A to produce a B. Finally, ($\oplus$ Intro-R) says that if $\Gamma \vdash A$, then $\Gamma \vdash A$ "or" B, but we're not necessarily able to produce B. Later, we're going to show how to interpret these rules in our categorical model, **DC**.

3 The Dialectica categories

3.1 The base category DC

Now that we understand the logic that we're trying to model, we will give a construction for the category **DC** from a category **C**. This construction is given in Chapter 1 of **de1991dialectica**. Given some category **C** with finite limits (importantly, finite products and pullbacks), we can construct a category **DC** as follows.

- The objects in **DC** are triples (U, X, α) where U and X are objects in **C**, and α is a relation on $U \times X$, where U and X are interpreted as sets. What this really means is that α is a *subobject* of the product $U \times X$, but we will stick to the relation terminology.
- The morphisms of **DC** are pairs of morphisms (f, F), where $f : U \to V$ is a "forward" map and $F : U \times Y \to X$ is a "backward" map. A morphism from (U, X, α) to (V, Y, β) must satisfy the following condition:
$$u\ \alpha\ F(u, y) \Rightarrow f(u)\ \beta\ y.$$
- The identity morphism on (U, X, α) is (id_U, π_2). (Check that it satisfies the condition!)
- The composition of morphisms $(f, F) : (U, X, \alpha) \to (V, Y, \beta)$ and $(g, G) : (V, Y, \beta) \to (W, Z, \gamma)$ is as follows:
  - The forward map $U \to W$ is the composition $g \circ f$.
  - The backward map $U \times Z \to X$ is composed as follows:
$$U \times Z \xrightarrow{\Delta \times \mathrm{id}_Z} U \times U \times Z \xrightarrow{\mathrm{id}_U \times f \times \mathrm{id}_Z} U \times V \times Z \xrightarrow{\mathrm{id}_U \times G} U \times Y \xrightarrow{F} X.$$
    Here, $\Delta : U \to U \times U$ is the diagonal morphism. So, the backward map is not as nice but it still works.

3.2 More constructions in DC

Given some extra conditions on **C**, the logical connectives we showed earlier from linear logic have their corresponding operations in **DC**:

- tensor product $\otimes$,
- categorical product &,
- internal Hom functor $\multimap$ satisfying the following natural isomorphism:

$$\mathrm{Hom}(\alpha \otimes \beta, \gamma) \cong \mathrm{Hom}(\alpha, \beta \multimap \gamma),$$

- and weak coproducts $\oplus$.

3.2.1 Tensor product

The tensor product $\alpha \otimes \beta$ of two relations (U, X, α) and (V, Y, β) is a relation $(U \times V, X \times Y, \otimes^\alpha_\beta)$ where

$$(u, v) \otimes^\alpha_\beta (x, y) \text{ if } u\ \alpha\ x \text{ and } v\ \beta\ y.$$

We can also easily define $(f, F) \otimes (g, G)$ as $(f \times g, F \times G)$, so $\otimes$ is a bifunctor.

The tensor product is not a categorical product because we don't know how to define projections. To define a projection

$$(U \times V, X \times Y, \otimes^\alpha_\beta) \to (U, X, \alpha),$$

say, we would need a backward map $F : (U \times V) \times X \to X \times Y$, and it's not clear how we would procure an element of Y. We similarly don't have a diagonal morphism

$$(U, X, \alpha) \to (U, X, \alpha) \otimes (U, X, \alpha).$$

We would need a backward map $F : U \times (X \times X) \to X$, and if we pick either canonical projection, we don't satisfy the condition on morphisms.

3.2.2 Categorical product

If **C** has stable and disjoint coproducts, we can define the categorical product $\alpha\ \&\ \beta$ as a relation $(U \times V, X + Y, \&^\alpha_\beta)$ where

$$(u, v)\ \&^\alpha_\beta\ z \text{ if } \begin{cases} u\ \alpha\ z & z \in X \\ v\ \beta\ z & z \in Y. \end{cases}$$

We *can* define projections for $\alpha\ \&\ \beta$. To define a projection

$$(U \times V, X + Y, \&^\alpha_\beta) \to (U, X, \alpha),$$

we can simply take the backward map $F : (U \times V) \times X \to X + Y$ to be the projection onto X then the inclusion into $X + Y$. We can also prove that the universal property holds for $\alpha\ \&\ \beta$.

3.2.3 Internal Hom functor

If **C** is locally Cartesian closed, given two relations (V, Y, β) and (W, Z, γ), we define their internal Hom object $\beta \multimap \gamma$ as a relation

$$(W^V \times Y^{V \times Z}, V \times Z, \multimap^\beta_\gamma)$$

which represents the set of morphisms between (V, Y, β) and (W, Z, γ). Intuitively, $\multimap^\beta_\gamma$ is a relation on potential morphisms (f, F) and elements $(v, z) \in V \times Z$ where

$$(f, F) \multimap^\beta_\gamma (v, z) \text{ if } v\ \beta\ F(v, z) \Rightarrow f(v)\ \gamma\ z.$$

Note that this relation encodes the condition on morphisms in **DC**.

3.2.4 Weak coproduct

Finally, we can define the weak coproduct $\alpha \oplus \beta$ as a relation $(U+V, X^U \times Y^V, \oplus^\alpha_\beta)$ where

$$w \oplus^\alpha_\beta (f,g) \text{ if } \begin{cases} w\ \alpha\ f(w) & w \in U \\ w\ \beta\ g(w) & w \in V. \end{cases}$$

We can easy define canonical injections into the weak coproduct.

4 Interpretation of linear logic into DC

First, fix an interpretation function $[\![\cdot]\!]$, which takes logical formulas to objects in **DC**. Looking back at our logical rules, Γ is a finite collection of logical formulas $A_1, \ldots, A_n$. We interpret Γ as:

$$[\![\Gamma]\!] = [\![A_1]\!] \otimes \cdots \otimes [\![A_n]\!].$$

Furthermore, this interpretation function maps the logical connectives to their corresponding operations in **DC**. We assume **C** has finite limits, stable and disjoint coproducts, and is locally Cartesian closed. Now, we can prove the following theorem:

Theorem 1. *If $\Gamma \vdash A$ in intuitionistic linear logic, then there exists a morphism in **DC** $(f,F) : [\![\Gamma]\!] \to [\![A]\!]$.*

Proof. We perform induction on the derivation of $\Gamma \vdash A$ using the inference rules of linear logic. We consider the last inference rule applied and show the cases for the four inference rules given above.

Case ($\otimes$ Intro). Suppose we applied the rule

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\ (\otimes\ \textsc{Intro})$$

By the inductive hypothesis, we have morphisms

$$(f,F) : [\![\Gamma]\!] \to [\![A]\!] \text{ and } (g,G) : [\![\Delta]\!] \to [\![B]\!].$$

As $\otimes$ is a bifunctor, we simply take our final morphism to be

$$(f,F) \otimes (g,G) : [\![\Gamma]\!] \otimes [\![\Delta]\!] \to [\![A]\!] \otimes [\![B]\!].$$

Case (& Intro) Suppose we applied the rule

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \ \&\ B}\ (\&\ \textsc{Intro})$$

By the inductive hypothesis, we have morphisms

$$(f,F) : [\![\Gamma]\!] \to [\![A]\!] \text{ and } (g,G) : [\![\Gamma]\!] \to [\![B]\!].$$

By the universal property of &, there exists a morphism

$$\langle (f,F), (g,G) \rangle : [\![\Gamma]\!] \to [\![A]\!] \ \&\ [\![B]\!].$$

Case ($\multimap$ Intro) Suppose we applied the rule

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\ (\multimap\ \textsc{Intro})$$

By the inductive hypothesis, we have a morphism

$$(f,F) : [\![\Gamma]\!] \otimes [\![A]\!] \to [\![B]\!].$$

As **DC** is monoidal closed with respect to the internal Hom functor $\multimap$, there exists a morphism

$$(g,G) : [\![\Gamma]\!] \to [\![A]\!] \multimap [\![B]\!].$$

Case (⊗ Intro-R) Suppose we applied the rule

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B}\ (\oplus\ \textsc{Intro-R})$$

By the inductive hypothesis, we have a morphism

$$(f, F) : [\![\Gamma]\!] \to [\![A]\!].$$

We take our final morphism to be

$$i_1 \circ (f, F) : [\![\Gamma]\!] \to [\![A]\!] \oplus [\![B]\!]$$

where i_1 is the canonical inclusion. □

So, what does it mean to have a categorical model of linear logic? Well, in general, having a model shows that a logic is consistent, in that you can't derive a contradiction from the logic. You can also use a model to prove that a particular formula is not derivable in a logic. A categorical model also captures proofs as morphisms. Now, we can look at proofs more closely—maybe two proofs are interpreted as the same morphism, so we can consider them the same proof. In our case, we're more interested in the **DC** category itself.

5 The Bean type system

5.1 Type systems

A type system is just like a deductive logic, except it tells you how to construct a valid *program* and give it a *type*. Rather than formulas A, we have variables x and y, and programs which use variables, like **add** x y. We also have *types*, like $\mathbb{R}$, which we give to variables and programs. In Bean, a context Γ is a set of variables and their types. Now, as an example, the *typing judgment*

$$\Gamma, x : \mathbb{R}, y : \mathbb{R} \vdash \textbf{add}\ x\ y : \mathbb{R}$$

means that if x and y are variables of type (are elements of) $\mathbb{R}$ in our context, then the program **add** x y also has type $\mathbb{R}$. We apply rules from our type system to derive such a judgment and prove our program is well-formed.

5.2 Backward error and Bean

The purpose of Bean is to write numerical programs and determine their *roundoff error*, also called floating-point error. This is the error that accumulates every time a computer rounds an exact answer to the nearest representable floating-point number. In Bean, we track a quantity known as *backward error*. Essentially, if f is a function, and $\tilde{f}$ is a program which approximates f, then the backward error of $\tilde{f}$ with respect to an input x is the difference between x and another input $\tilde{x}$ (known as the witness) such that $\tilde{f}(x) = f(\tilde{x})$. We use *annotations* in the context to keep track of this backward error. For example, here is one of Bean's typing rules:

$$\frac{}{\Gamma, x :_{\varepsilon} \mathbb{R}, y :_{\varepsilon} \mathbb{R} \vdash \textbf{add}\ x\ y : \mathbb{R}}\ (\textsc{Add})$$

It says that for the program **add** x y, the backward error with respect to x and y is ε, where ε is a small quantity known as *unit roundoff* (specific to a machine's floating-point format). Thus, there exist real numbers $\tilde{x}, \tilde{y}$, where the difference between x and $\tilde{x}$ and between y and $\tilde{y}$ is at most ε, such that

$$\textbf{add}\ x\ y = \tilde{x} + \tilde{y}.$$

The smaller the backward error with respect to its inputs, the better an approximation the program. The known backward errors for these basic operations is built in to Bean's type system, but the novelty is that we can compose several of Bean's rules to determine the backward error for larger programs, such as:

$$\dfrac{\dfrac{}{x :_\varepsilon \mathbb{R}, y :_\varepsilon \mathbb{R} \vdash \textbf{add}\ x\ y} \qquad \dfrac{}{a :_\varepsilon \mathbb{R}, b :_\varepsilon \mathbb{R} \vdash \textbf{mul}\ x\ y}}{x :_{2\varepsilon} \mathbb{R}, y :_{2\varepsilon} \mathbb{R}, b :_\varepsilon \mathbb{R} \vdash \textbf{let}\ a = \textbf{add}\ x\ y\ \textbf{in}\ \textbf{mul}\ a\ b}$$

The program **let** $a =$ **add** $x\ y$ **in mul** $a\ b$ approximates the function $(x + y) \cdot b$ for real numbers x, y, b. Notice how the error accumulated for x and y to 2ε, since they were calculated earlier on in the program.

5.3 Linearity in Bean

Bean is a *linear* type system, a term inspired by linear logic. In particular, context variables can't appear twice in a program. This corresponds to the fact that we can't reuse or duplicate contexts in linear logic. In the above example program, x, y, and b were each used exactly once in the program.

This restriction is due to the nature of backward error. If f and g both have some backward error with respect to x, then we can find an $\tilde{x}_f$ and an $\tilde{x}_g$ such that $f(\tilde{x}_f) = \tilde{f}(x)$ and $g(\tilde{x}_g) = \tilde{g}(x)$. However, if we wanted to approximate the function $f + g$ with $\tilde{f} + \tilde{g}$, we may not be able to find a *single* $\tilde{x}$ such that

$$f(\tilde{x}) + g(\tilde{x}) = \tilde{g}(x) + \tilde{f}(x),$$

if $\tilde{x}_f \neq \tilde{x}_g$. Hence, $f(x) + g(x)$ is not guaranteed to have a backward error witness. In Bean, therefore, we don't allow programs approximating $f + g$ and the like by disallowing duplicate variables.

6 The Bel category

Now, we're going to switch gears and talk about the category, **Bel**, which gives the semantics of the Bean type system.

- The objects of **Bel** are triples (X, d_X, r_X) where X is a set, $d_X : X \times X \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ is a function giving a notion of distance, and $r_X \in \mathbb{R}_{\geq 0} \cup \{\infty\}$ is a constant called the *slack*.
  - This doesn't look too much like a relation right now, but we could alternatively write (X, d_X, r_X) as $\{R_\delta : \delta \in \mathbb{R}_{\geq 0}\}$ where $x\ R_\delta\ y$ if $d_X(x, y) - r_X \leq \delta$.
- Morphisms between (X, d_X, r_X) and (Y, d_Y, r_Y) are triples of functions $(f, \tilde{f}, b)$ where $f, \tilde{f} : X \to Y$ and $b : X \times Y \rightharpoonup X$. For $x \in X$ and $y \in Y$ such that $d_Y(\tilde{f}(x), y) < \infty$, these functions must satisfy the following conditions:
  1. $d_X(x, b(x, y)) - r_X \leq d_Y(\tilde{f}(x), y) - d_Y$, and
  2. $f(b(x, y)) = y$.

  The idea is that f represents an ideal function, $\tilde{f}$ represents our approximating program, and b maps inputs and approximate outputs to a backward error witness, i.e. maps x and $\tilde{f}(x)$ to an $\tilde{x}$. The slack becomes important later for interpreting a backward error bound.
- The identity morphism for (X, d_X, r_X) is $(\mathrm{id}_X, \mathrm{id}_X, \pi_2)$.
- Given two morphisms $(f_1, \tilde{f}_1, b_1)$ and $(f_2, \tilde{f}_2, b_2)$, their composition is very similar to that in **DC**. The composition of our forward maps is straightforward composition, but our backward map composition takes (x, z) to $b_1(x, b_2(\tilde{f}_1(x), z))$.

In **Bel**, we can define a symmetric monoidal product $\otimes$, but not a categorical product because we can't, as in **DC**, define a diagonal map $\Delta : X \to X \otimes X$. This is because we would need a backward map $b : X \times X \to X$, but if we choose either projection, we won't be able to satisfy the second condition on morphisms. We do have projections in some limited cases as well as coproducts.

Bel has a graded comonad, D_r, which takes an object (X, d_X, r_X) to $(X, d_X, r_X + r)$. It is the identity on morphisms. We will use this graded comonad to interpret the backward error annotations from Bean.

7 Interpretation of Bean into Bel

A well-typed program in Bean corresponds to a morphism in **Bel**. First, we interpret Bean types as **Bel** objects, e.g. $[\![\mathbb{R}]\!] = (\mathbb{R}, d, 0)$ and d is a metric on $\mathbb{R}$. Next, we interpret contexts as

$$[\![\Gamma, x :_r \mathbb{R}]\!] = [\![\Gamma]\!] \otimes D_r[\![\mathbb{R}]\!].$$

Theorem 2. *If $\Gamma \vdash e : \sigma$ is a* Bean *program (where σ is a type), then we can build a morphism from $[\![\Gamma]\!] \to [\![\sigma]\!]$.*

We do so using compositions of various canonical maps and by induction on the Bean typing derivation. Now, we can use this categorical model to prove that Bean programs satisfy the backward error condition. For example, if $x :_\varepsilon \mathbb{R}, y :_\varepsilon \mathbb{R} \vdash$ **add** $x\ y : \mathbb{R}$ is our program, we can build a morphism

$$\begin{aligned}(f, \tilde{f}, b) &: D_\varepsilon(\mathbb{R}, d, 0) \otimes D_\varepsilon(\mathbb{R}, d, 0) \to (\mathbb{R}, d, 0)\\ &= (\mathbb{R} \times \mathbb{R}, d_\otimes, \varepsilon) \to (\mathbb{R}, d, 0).\end{aligned}$$

These maps must satisfy the conditions on morphisms in Bean. Remember, $f(x, y)$ represents our ideal function $x + y$ and $\tilde{f}(x, y)$ represents our approximating program **add** $x\ y$. Now, given $x, y \in \mathbb{R}$, set

$$(\tilde{x}, \tilde{y}) = b(x, y, \tilde{f}(x, y)).$$

These are our backward error witnesses because

$$f(\tilde{x}, \tilde{y}) = \tilde{f}(x, y)$$

by condition 2, and

$$d_\otimes((x, y), (\tilde{x}, \tilde{y})) - \varepsilon \leq d(\tilde{f}(x, y), \tilde{f}(x, y)) = 0$$

by condition 1 which, by definition of $d_\otimes$, means

$$d(x, \tilde{x}) \leq \varepsilon \text{ and } d(y, \tilde{y}) \leq \varepsilon.$$

Therefore, $\tilde{x}$ and $\tilde{y}$ are the backward error witnesses we needed to prove that the backward error of $\tilde{f}(x, y)$ with respect to x and y is ε.

8 Goals

8.1 Function spaces

In a functional programming language, we ideally want to be able to write, well, functions. TODO

8.2 Probabilistic programs

Can we think of idealized and approximate functions as probabilistic programs in our language? This would entail adding some distribution monad as part of our composition, for which we can use the coend calculus **loregian2021co**. The goal is to maintain backwards error bounds even under randomized computation. To do this, we can treat these programs as effectful programs by attaching a monad to the lens-maps.

9 Conclusion