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*.

- Introduced by de Paiva [1], the Dialectica categories are categorical models of intuitionistic linear logic. From a base category **C**, we can construct a Dialectica category, abbreviated **DC**, such that logical formulas are interpreted as objects in **DC** and judgments are interpreted as morphisms.
- The objects of **DC** are internal binary *relations* on **C**.
- The morphisms in **DC** are pairs of morphisms in **C** satisfying a certain condition. These morphisms are 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**, introduced by Kellison et al. [2].

- **Bel** is the category used to interpret a linear type system, Bean, which derives bounds on roundoff error in numerical programs. We show soundness of these error bounds by interpreting Bean typing judgments as morphisms in **Bel**.
- The objects of **Bel** are similar to metric spaces, and 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

The Dialectica categories model (intuitionistic) *linear logic*, which we will define as a system of natural deduction [4]. Essentially, we prove a *judgment* by applying *inference rules* until we reach *axioms*, or judgments we are allowed to take for granted. The judgment

$$\Gamma \vdash A$$

means "given the formulas in Γ, we can prove A," where A is a logical *formula* and $\Gamma = A_1, \ldots, A_n$ is a list of formulas (our *context*). If Γ contains A, this judgment is trivial, and is in fact an axiom.

Linear logic is a *substructural* logic, which means it's missing some of the common inference rules found in other logics. Specifically, linear logic is missing *weakening* and *contraction*.

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

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 consume the resources in Δ. If $\Gamma, A, A \vdash B$, then B consumes exactly two copies of A, and we cannot conclude $\Gamma, A \vdash B$. Linear logic is nice because it can model real-world situations, like chemical reactions and concurrent processes.

To get a sense of linear logic, we display here four of its important inference rules:

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

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

The formula $A \otimes B$ asserts that we have the resources to produce both A and B. The formula $A \,\&\, B$ also means we have the resources to produce both A and 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, $A \multimap B$ is logical implication, so if we are given an A, we can produce a B. Finally, $A \oplus B$ asserts we can produce one of A or B, but we don't necessarily know which one. 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 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 de Paiva [1]. Given some category **C** with finite limits, 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 subobject of the product $U \times X$.

  Remark 1. *The intuition, which we use throughout this blog post, is that we take* ***C*** *to be* ***Set****, and α to be a* binary relation *on the set $U \times X$. Thus, we write $u \;\alpha\; x$ to mean (u, x) is in the relation $\alpha \subseteq U \times X$.*

- The morphisms of **DC** are pairs of morphisms (f, F), where $f : U \to V$ is called the "forward" map and $F : U \times Y \to X$ is called the "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 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.

3.2 More constructions in DC

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

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

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

- and weak coproducts $\oplus$.

We show constructions for the first three.

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 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 wouldn't satisfy the condition on morphisms.

3.2.2 Categorical product

If $\mathbf{C}$ 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 of products holds for $\alpha \ \& \ \beta$.

3.2.3 Internal Hom functor

If $\mathbf{C}$ 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.

4 Interpretation of linear logic into DC

Now, we are ready to show that $\mathbf{DC}$ is a categorical model of linear logic.

4.1 Inductive construction

First, fix an interpretation function $[\![\cdot]\!]$ which takes atomic propositions to objects in $\mathbf{DC}$. The interpretation function is then extended to formulas and maps the logical connectives to their corresponding constructions in $\mathbf{DC}$. A context $\Gamma = A_1, \ldots, A_n$ is interpreted as

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

Now, we can prove the following theorem:

Theorem 1. *If we can derive $\Gamma \vdash A$ in linear logic, then we can build a morphism in* ***DC*** *from $[\![\Gamma]\!] \to [\![A]\!]$.*

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 a representative case.

Case (& Intro) Suppose we applied the rule

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

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

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

4.2 Discussion

What does it mean to have a categorical model of linear logic? In general, having a model shows that a logic is consistent, in that you can't derive a contradiction from the inference rules. You can also use a model to prove that a particular formula is not derivable in a logic. A categorical model is especially useful as it 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, however, we're more interested in the construction of **DC** itself. As we briefly mentioned in the introduction, we are developing **Bel**, the category of backward error lenses, which resembles **DC** in many ways. A lens is essentially a pair of morphisms, "get" and "put," much like the forward and backward maps of **DC** [3]. In **Bel**, as in most lens categories, it is unclear how to define products and the internal Hom functor. Thus, we hope that by reframing **Bel** to look like **DC**, we can reproduce these constructions.

5 The Bean type system

Now, we switch gears and talk about the Bean type system, for which **Bel** serves as a categorical model. Both Bean and **Bel** were developed by Kellison et al. [2]. We will explore the ways that Bean resembles linear logic.

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 programs which use variables, like **add** x y. We also have types τ which we give to variables and programs using a colon, like $x : \tau$. Our base type is $\mathbb{R}$, which represents the real numbers, but we can form more complex types like $\mathbb{R} \otimes \mathbb{R}$, representing $\mathbb{R}^2$.

A context $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ is a finite set of variables and their types. 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 $\mathbb{R}$ 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. Additionally, given a Bean program, we want to be able to algorithmically come up with its typing derivation, a process known as *type checking*.

5.2 Backward error and Bean

Bean determines *roundoff error*, also called floating-point error, for numerical programs. This is the error that accumulates every time a computer rounds an exact answer to the nearest representable floating-point number. In Bean, we bound a quantity known as *backward error*.

Definition 1. *Let f be a function with input x and $\tilde{f}$ a program which approximates f. The backward error of $\tilde{f}$ with respect to x is the distance $d(x, \tilde{x})$ 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:

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

It says that for the program **add** x y, the backward error with respect to x and y is bounded by ε, where ε is a small number known as *unit roundoff* (specific to a computer's floating-point format). Thus, there exist real numbers $\tilde{x}, \tilde{y}$, where $d(x, \tilde{x}), d(y, \tilde{y}) \leq \varepsilon$, such that

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

This result is a basic assumption for numerical error analysis.

The smaller the backward error with respect to its inputs, the better an approximation the program. The known backward errors for 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:

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

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 backward 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. It means that 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

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

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 category Bel

Now, we're going to introduce the category, **Bel**, which gives the categorical semantics of the Bean type system.

- The objects of **Bel** are triples (X, d, r) where
  - X is a set,
  - $d : X \times X \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ is a function giving a notion of distance,
  - and $r \in \mathbb{R}_{\geq 0} \cup \{\infty\}$ is a constant called the *slack*.
- 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 $b : X \times Y \to X$. For $x \in X$ and $y \in Y$, 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 our exact mathematical 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.

Remark 2. *A morphism in **Bel** is therefore known as a* backward error lens*, where f and $\tilde{f}$ are our "get" maps and b is our "put" map. The two conditions are known as the first and second lens conditions.*

- The identity morphism for (X, d, r) 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 ordinary function 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_p, which takes an object (X, d, r) to $(X, d, r + p)$. 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

7.1 Inductive construction

A well-typed Bean program corresponds to a morphism in **Bel**. First, we interpret Bean types as **Bel** objects, e.g. $[\![\mathbb{R}]\!] = (\mathbb{R}, d, 0)$ where 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 we can derive $\Gamma \vdash e : \tau$ in* Bean*, then we can build a morphism in **Bel** from $[\![\Gamma]\!] \to [\![\tau]\!]$.*

Proof. Just as we did for linear logic and **DC**, we perform induction on the Bean typing derivation and use compositions of various canonical maps. □

7.2 Backward error soundness theorem

Our categorical model serves a specific purpose: to prove that well-typed Bean programs satisfy the corresponding backward error guarantee. For demonstration, we use the simple program $x :_\varepsilon \mathbb{R}, y :_\varepsilon \mathbb{R} \vdash \mathbf{add}\ x\ y : \mathbb{R}$, but this proof can be generalized to all Bean programs.

Theorem 3. *If we can derive $x :_\varepsilon \mathbb{R}, y :_\varepsilon \mathbb{R} \vdash \mathbf{add}\ x\ y : \mathbb{R}$ in* Bean*, then for all inputs $x, y \in \mathbb{R}$, we can find backward error witnesses $\tilde{x}, \tilde{y}$ such that $\mathbf{add}\ x\ y = \tilde{x} + \tilde{y}$ and $d(x, \tilde{x}), d(y, \tilde{y}) \leq \varepsilon$.*

Proof. As our program $x :_\varepsilon \mathbb{R}, y :_\varepsilon \mathbb{R} \vdash \mathbf{add}\ x\ y : \mathbb{R}$ is well-typed, we can follow the proof of Theorem 2 to build a backward error lens:

$$(f, \tilde{f}, b) : (\mathbb{R} \times \mathbb{R}, d, \varepsilon) \to (\mathbb{R}, d, 0).$$

By construction, we have $f(x, y) = x + y$ and $\tilde{f}(x, y) = \mathbf{add}\ x\ y$. Moreover, these maps satisfy the lens conditions. Given inputs $x, y \in \mathbb{R}$, set

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

These are our backward error witnesses because

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

by the second lens condition, and

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

by the first lens condition. By definition of d, this means

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

Therefore, $\tilde{x}$ and $\tilde{y}$ are our desired backward error witnesses. □

8 Conclusion

There are many resemblances between **DC**, a categorical model of linear logic, and **Bel**, the category of backward error lenses. Moreover, they each interpret a linear deductive system. The categories may not look very similar at first glance, but we are investigating whether **Bel** could be a full subcategory of **DC** (where **C** is **Set**).

We can do so by rewriting a **Bel** object (X,d,r) as $(X,X,\{\alpha_i\})$ where $x\ \alpha_i\ y$ if $d(x,y) - r > i$. This is not exactly a **DC** object, as we have a family of relations rather than a single relation, but we simply need to rewrite the **DC** condition as

$$\forall i,\ u\ \alpha_i\ F(u,y) \Rightarrow f(u)\ \beta_i\ y$$

and the categorical constructions still hold. Moreover, given two **DC** objects whose relations correspond to metrics, from the modified **DC** condition we can recover the first **Bel** lens condition, where our forward map is the approximate map $\tilde{f}$.

Now, we need only embed the second **Bel** lens condition into **DC**. One possibility is by requiring that the backward map is injective in its second component. This would allow us to recover an exact **Bel** map from a **DC** morphism.

Once we find such an embedding, we can work directly with constructions in **DC**, allowing us to enrich the Bean type system. For example, we would immediately be able to define higher-order functions in Bean, as we would have internal Homs in **Bel**. With a little more work, we could define monads for modeling effects. In any case, the deep correspondence between **DC** and **Bel** is remarkable and unexpected.

References

[1] Valeria Correa Vaz De Paiva. *The dialectica categories*. Tech. rep. University of Cambridge, Computer Laboratory, 1991.

[2] Ariel E Kellison et al. "Bean: A Language for Backward Error Analysis". In: *arXiv preprint arXiv:2501.14550* (2025).

[3] nLab authors. *lens*. `https://ncatlab.org/nlab/show/lens+%28in+computer+science%29`. Revision 44. May 2025.

[4] nLab authors. *natural deduction*. `https://ncatlab.org/nlab/show/natural+deduction`. Revision 43. May 2025.