Formalizing Typing Rules for Natural Languages using Effects

Matthieu Boyer

19 mars 2025

Table des matières

1 Basic Idea

The main idea is that you can consider some words to act as functors, for example determiners : they don't change the way a word acts in a sentence, but more the setting in which the word works.

2 Formally

In the next section, we will designate by $\mathcal{L}$ our language, as a set of words (with their semantics) and syntactic rules to combine them semantically. We denote by $\mathcal{O}(\mathcal{L})$ the set of words in the language whose semantic representation is a low-order function and $\mathcal{F}(\mathcal{L})$ the set of words whose semantic representation is a functor or high-order function. In the following, we will consider the different ways we can represent algebraically our typing rules.

2.1 Polymorphism

Let $\mathcal{C}$ be a closed cartesian category. This represents our main typing system, consisting of words $\mathcal{O}(\mathcal{L})$ that can be expressed without effects. Remember that $\mathcal{C}$ contains a terminal object $\perp$ representing the empty type or the lack thereof. We can consider $\bar{\mathcal{C}}$ the category closure of $\mathcal{F}(\mathcal{L})(\mathcal{O}(\mathcal{L}))$, that is consisting of all the different type constructors (ergo, functors) that could be formed in the language. In that sense, $\bar{\mathcal{C}}$ is still a closed cartesian category (since all our functors induce isomorphisms on their image) [1]. $\bar{\mathcal{C}}$ is our typing category.

We consider for our types the quotient set $\star = \mathrm{Obj}\left(\bar{\mathcal{C}}\right)/\mathcal{F}(\mathcal{L})$. Since $\mathcal{F}(\mathcal{L})$ does not induce an equivalence relation on $\mathrm{Obj}\left(\bar{\mathcal{C}}\right)$ but a preorder, we consider the chains obtained by the closure of the relation $x \succeq y \Leftrightarrow \exists F, y = F(x)$ (which

1. We can even close it if we want to.

is seen as a subtyping relation). We also define $\star_0$ to be the set obtained when considering types which have not yet been *affected*, that is $\mathrm{Obj}(\mathcal{C})$. In contexts of polymorphism, we identify $\star_0$ to the adequate subset of $\star$.

In this paradigm, constant objects (or results of fully done computations) are functions with type $\bot \to \tau$ which we will denote directly by $\tau \in \star_0$.

This leads us to consider functors as polymorphic functions : for a (possibly restrained, though it seems to always be $\star$) set of base types S, a functor is a function

$$x : \tau \in S \subseteq \star \mapsto Fx : F\tau$$

Remember that $\star$ is a fibration of the types in $\bar{\mathcal{C}}$. This means that if a functor can be applied to a type, it can also be applied to all *affected* versions of that type, i.e. $\mathcal{F}(L)(\tau \in \star)$. More importantly, while it seems that F's type is the identity on $\star$, the important part is that it changes the effects applied to x (or τ). In that sense, F has the following typing judgements :

$$\frac{\Gamma \vdash x : \tau \in \star_0}{\Gamma \vdash Fx : F\tau \notin \star_0}\ \mathrm{Func}_0 \qquad \frac{\Gamma \vdash x : \tau}{\Gamma \vdash Fx : F\tau \preceq \tau}\ \mathrm{Func}$$

We use the same notation for the *language functor* and the *type functor* in the examples, but it is important to note those are two different objects, although connected.

In the same fashion, we can consider functions (with a non-zero arity) to have a type in $\star$ or more precisely of the form $\star \to \star$ which is a subset of $\star$. This explains how functors can act on functions.

In that sense, applicatives and monads only provide with more flexibility on the ways to combine functions : they provide intermediate judgements to help with the combination of trees. For example, the multiplication of the monad :

$$\frac{\Gamma \vdash x : MM\tau}{\Gamma \vdash x : M\tau \preceq MM\tau}\ \mathrm{Monad}$$

The difference between this judgement and the judgement Func is that here we have equality between the two types $MM\tau$ and $M\tau$.

We could also add judgements for adjunctions, but the most interesting thing is to add judgements for natural transformations. While in general we do not want to find natural transformations, we want to be able to express these in two situations :

1. If we have an adjunction $L \dashv R$, we have natural transformations for $\mathrm{Id}_{\mathcal{C}} \Rightarrow L \circ R$ and $R \circ L \Rightarrow \mathrm{Id}_{\mathcal{C}}$.
2. Though I don't have an example in the english language yet, there could be words which act as functor modificators or transformations, which will probably be natural. In the event where such transformations exist, we will be able to express them easily.

To complete this section, let's look at a simple list of different typing composition judgements through which we also re-derive the subtyping judgement to allow for its implementation. Note that here, the syntax is not taken into account : a function is always written left of its arguments, whether or not they are actually in that order in the sentence. This issue will be resolved by giving the syntactic tree of the sentence (or infering it at runtime).

2.2 Product Category (and a bit of polymorphism)

Another way to start would be to consider product categories : one for the main type system and one for the effects. Let $\mathcal{C}_0$ be a closed cartesian category representing our main type system. Here we again consider constants and full computations as functions $\bot \to \tau$ or $\tau \in \mathrm{Obj}(\mathcal{C}_0)$. Now, to type functions and functors, we need to consider a second category : We consider $\mathcal{C}_1$ the category representing the free monoid on $\mathcal{F}(\mathcal{L})$. Monads and Applicatives will generate relations in that monoid. To ease notation we will denote *functor types* in $\mathcal{C}_1$ as lists written with head on the left.

Finally, let $\mathcal{C} = \mathcal{C}_0 \times \mathcal{C}_1$ be the product category. This will be our typing category. This means that the real type of objects will be $(\bot \to \tau, [])$, which we will still denote by τ. We will denote by $F_n \cdots F_0\tau$ the type of an object, as if it

$$\forall F \in \mathcal{F}(\mathcal{L}), \qquad \frac{\Gamma \vdash x : \tau \qquad \Gamma \vdash F : S \subseteq \star \qquad \overbrace{\tau \in S}^{\exists \tau' \in S, \tau \preceq \tau'}}{\Gamma \vdash Fx : F\tau \preceq \tau} \text{ Cons}$$

$$\frac{\Gamma \vdash x : \tau \qquad \tau \in \star_0}{\Gamma \vdash Fx : F\tau \notin \star_0} \; FT_0$$

$$\frac{\Gamma \vdash x : F\tau_1 \qquad \Gamma \vdash \varphi : \tau_1 \to \tau_2}{\Gamma \vdash \varphi x : F\tau_2} \; \texttt{fmap}$$

$$\frac{\Gamma \vdash x : \tau_1 \qquad \Gamma \vdash \varphi : \tau_1 \to \tau_2}{\Gamma \vdash \varphi x : \tau_2} \text{ App}$$

$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash x : A\tau} \; \texttt{pure/return}$$

$$\frac{\Gamma \vdash x : A\tau_1 \qquad \Gamma \vdash \varphi : A(\tau_1 \to \tau_2) = A\tau_1 \to A\tau_2}{\Gamma \vdash \varphi x : A\tau_2} \; \texttt{<*>}$$

$$\frac{\Gamma \vdash x : MM\tau}{\Gamma \vdash x : M\tau} \; \texttt{>>=}$$

$$\forall F \overset{\theta}{\Longrightarrow} G, \qquad \frac{\Gamma \vdash F : S \subseteq \star \qquad \Gamma \vdash x : \tau \in S \cap S' \qquad \Gamma \vdash G : S' \subseteq \star \qquad \Gamma \vdash \varphi : \tau \to \tau'}{\Gamma \vdash F\varphi Fx = F(\varphi x) : G\tau'} \; \texttt{nat}$$

TABLE 1 – Typing and Subtyping Judgements

were a composition of functions[2].

In that paradigm, functors simply append to the head of the *functor type* (with the same possible restrictions as before, though I do not see what they would be needed for) while functions will take a polymorphic form : $x : L\tau_1 \mapsto \varphi x : L\tau_2$ and φ's type can be written as $\star\tau_1 \to \star\tau_2$.

2. It is!