Formalizing Typing Rules for Natural Languages using Effects

Matthieu Boyer

March 24, 2025

Contents

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. This document is based on [BC].

2 Typing System

In this 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.

2.1   Typing Category

2.1.1   Types

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 is seen as a subtyping relation as proposed in [MZ15]). 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 $\perp \to \tau$ which we will denote directly by $\tau \in \star_0$.

2.1.2   Functors, Applicatives and Monads

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:

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

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:

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

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

2.1.3   Natural Transformations for Handlers

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]We can even close it if we want to.

1. If we have an adjunction $L \dashv R$, we have natural transformations for $\text{Id}_{\mathcal{C}} \Rightarrow L \circ R$ and $R \circ L \Rightarrow \text{Id}_{\mathcal{C}}$.

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 see why we want this rule, which is a larger version of the monad multiplication and the monad/applicative unit, it suffices to see that the diagram below provides a way to construct the "correct function" on the "correct functor" side of types. If we have a natural transformation $F \overset{\theta}{\Longrightarrow} G$ then for all arrows $f : \tau_1 \to \tau_2$ we have:

$$
\begin{array}{ccc}
F\tau_1 & \overset{Ff}{\longrightarrow} & F\tau_2 \\
\theta_{\tau_1} \downarrow & & \downarrow \theta_{\tau_2} \\
G\tau_1 & \overset{Gf}{\longrightarrow} & G\tau_2
\end{array}
$$

and this implies, from it being true for all arrows, that from $\Gamma \vdash x : F\tau$ we have an easy construct to show $\Gamma \vdash x : G\tau$. This could also give us a way to construct handlers for our effects as per [BP14] or [PP13].

2.1.4   Natural Transformations for Higher-Order Constructs

We might want to add plurals, superlatives, tenses, aspects and other similar constructs which act as function modificators. For each of these, we give a functor Π corresponding to a new class of types along with natural transformations for all other functors F from F to $\Pi \circ F$ and $F \circ \Pi$ as well as from $\Pi \circ F$ to $F \circ \Pi$ which should be defined in a similar way for all functors. This allows us to add complexity not in the compositional aspects but in the lexicon aspects. We do not want these functors to be applicatives, as we do not want a way to *create* them in the parser if they are not marked in the sentence.

One of the main issues with this is the following: In the English language, plural is marked on all words (except verbs, and even then case could be made for it to be marked), while future is marked only on verbs (through the *will + verb* construct which creates a "*new*" verb in a sense) though it applies also to the arguments on the verb. A way to solve this would be to include in the natural transformations rules to propagate the functor depending on the type of the object. Consider the superlative effect **most**[2]. As it can only be applied on adjectives, we can assume its argument is a function (but the definition would hold anyway taking $\tau_1 = \bot$). It is associated with the following function (which is a rewriting of the natural transformation rule):

$$
\frac{\Gamma \vdash x : \tau_1 \to \tau_2}{\Gamma \vdash \mathbf{most}\, x := \Pi_{\tau_2} \circ x = x \circ \Pi_{\tau_1}}
$$

What curryfication implies, is that higher-order constructs can be passed down to the arguments of the functions they are applied to, explaining how we can reconciliate the following semantic equation even if some of the words are not marked properly:

$$
\mathbf{future}\,(\mathbf{be}\,(\mathbf{I}, \mathbf{a\ cat})) = \mathbf{will\ be}\,(\mathbf{future}\,(\mathbf{I}), \mathbf{a\ cat}) = \mathbf{be}\,(\mathbf{future}\,(\mathbf{I}), \mathbf{a\ cat})
$$

Indeed the above equation could be simply written by our natural transformation rule as:

$$
\mathbf{future}\,(\mathbf{be})\,(\mathbf{arg_1}, \mathbf{arg_2}) = \mathbf{future}\,(\mathbf{be})\,(\mathbf{arg_2})\,(\mathbf{future}\,(\mathbf{arg_1})) =
$$

2.2   Typing Judgements

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:

---

[2]We do not care about morphological markings here, we say that **largest** = **most** $\circ$ **large**

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

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

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

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

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

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

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

$$\forall F \stackrel{\theta}{\Longrightarrow} G, \qquad \frac{\Gamma \vdash x : F\tau \qquad \Gamma \vdash G : S' \subseteq \star \qquad \tau \in S'}{\Gamma \vdash x : G\tau} \quad \texttt{nat}$$

Table 1: Typing and Subtyping Judgements

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). We could also add symmetry induced rules for application.

3 Language Translation

In this section we will give a list of words along with a way to express them as either arrows or endo-functors of our typing category. This will also give a set of functors and constructs in our language.

3.1 Types of Syntax

For syntactic categories, the types that are generally used are the following, and we will see it matches with our lexicon, and simplifies our functorial definitions[3]. Those are based on [Par]. Here Υ is the operator which retrieves the type from a certain syntactic category.

---

[3]We don't consider effects in the given typings.

S t

CN(P) $e \to t$

ProperN e

NP Either:

e *John, a cat*

$\left(e \to t\right) \to t$ *the   cat,   every man*

DET $\Upsilon(\mathbf{CN}) \to \Upsilon(\mathbf{NP})$

ADJ(P) Either:

$\left(e \to t\right)$ *carnivorous, happy*

$\left(e \to t\right) \to \left(e \to t\right)$ *skillful*

REL $e \to t$

VP, IV(P) $e \to t$

TV(P) $\Upsilon(\mathbf{NP}) \to \Upsilon(\mathbf{VP})$

is $\left(e \to t\right) \to e \to t$

Table 2: Usual Typings for some Syntactic Categories

3.2   Lexicon: Semantic Denotations for Words

Many words will have basically the same "high-level" denotation. For example, the denotation for most common nouns will be of the form: $\Gamma \vdash \lambda x.\mathbf{planet}\,x : e \to t$. In Table 3 we give a lexicon for a subset of the english language. We describe the constructor for the functors used by our denotations in the table, but all functors will be reminded and properly defined in Table 4 along with their respective `fmap`.

Expression	Type	λ-Term
planet	$e \to t$	$\lambda x.\mathbf{planet}\,x$
	Similar to: **dog, cat, chair...**	
she	$e \to e$	$\lambda x.x$
which	$\left(e \to t\right) \to \mathtt{S}e$	$\lambda p.\{x \mid px\}$
the	$\left(e \to t\right) \to \mathtt{M}e$	$\lambda p.x$ if $p = \{x\}$ else $\#$
no	$\left(e \to t\right) \to \mathtt{C}e$	$\lambda p.\lambda c.\neg\exists x.px \wedge c\,x$
a	$\left(e \to t\right) \to \mathtt{D}e$	$\lambda s.\{\langle x, x + s\rangle \mid px\}$
it	$\left(\bot \to \mathtt{G}e\right)$	$\lambda g.g_0$

Table 3: λ-calculus representation of the english language $\mathcal{L}$

For most of the applicatives/monads in Table 4 we do not specify the unit and multiplication functions, as they are quite usual examples. We still provide the `fmap` for good measure.

Functor	Constructor	`fmap`	Typeclass
`G`	$G(\tau) = \mathtt{r} \to \tau$	$G\varphi(x) = \lambda r.\varphi(xr)$	Monad
`W`	$W(\tau) = \tau \times \mathtt{t}$	$W\varphi((a,p)) = (\varphi a, p)$	Monad
`S`	$S(\tau) = \{\tau\}$	$S\varphi(\{x\}) = \{\varphi(x)\}$	Monad
`M`	$M(\tau) = \tau + \bot$	$M\varphi(x) = \begin{cases} \varphi(x) & \text{if } \Gamma \vdash x : \tau \\ \# & \text{if } \Gamma \vdash x : \# \end{cases}$	Monad
`C`	$C(\tau) = \left(\tau \to \mathtt{t}\right) \to \mathtt{t}$	$C\varphi(x) = \lambda c.x\left(\lambda a.c\left(\varphi a\right)\right)$	Monad

Table 4: Denotations for the functors used

Let us explain a few of those functors: `G` encodes the possibility of reading from a certain environment of type `r` while `W` encodes the possibility of logging a message of type `t` with the expression.

A   Other Considered Things

A.1   Typing with a 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 were a composition of functions[4].

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 $\star\tau_1 \to \star\tau_2$.

B   Bibliography

[BC]   Dylan Bumford and Simon Charlow. Effect-driven interpretation.

[BP14] Andrej Bauer and Matija Pretnar. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science*, Volume 10, Issue 4, December 2014.

[MZ15] Paul-André Melliès and Noam Zeilberger. Functors are Type Refinement Systems. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–16, Mumbai India, January 2015. ACM.

[Par]   B Partee. Lecture 2. Lambda abstraction, NP semantics, and a Fragment of English. *Formal Semantics*.

[PP13] Gordon D. Plotkin and Matija Pretnar. Handling Algebraic Effects. *Logical Methods in Computer Science*, Volume 9, Issue 4, December 2013.

---

[4]It is!