

Formalizing Typing Rules for Natural Languages using Effects

Matthieu Boyer

March 25, 2025



Contents

1	Typing System	2
1.1	Typing Category	2
1.1.1	Types	2
1.1.2	Functors, Applicatives and Monads	2
1.1.3	Natural Transformations for Handlers	3
1.1.4	Natural Transformations for Higher-Order Constructs	3
1.2	Typing Judgements	4
2	Language Translation	5
2.1	Types of Syntax	5
2.2	Lexicon: Semantic Denotations for Words	5
2.3	Modality, Plurality, Aspect, Tense	7
2.3.1	An Example: The <i>Plural</i>	7
A	Other Considered Things	8
A.1	Typing with a Product Category (and a bit of polymorphism)	8
B	Bibliography	8

Abstract

The main idea is that you can consider some words to act as functors in a typing category, for example determiners: they don't change the way a word acts in a sentence, but more the setting in which the word works by adding an effect to the work. This document is based mostly on Bumford and Charlow and Bumford.

Introduction

Moggi (1989) provided a way to view monads as effects in functional programming. This allows for new modes of combination in a compositional semantic formalism, and provides a way to model words which usually raise problems with the traditional lambda-calculus representation of the words. In particular we consider words such as *the* or *a*

whose application to common nouns results in types that should be used in similar situations but with largely different semantics, and model those as functions whose application yields an effect. This allows us to develop typing judgements and an extended typing system for compositional semantics of natural languages. Type-driven compositional semantics acts under the premise that given a set of words and their denotations, a set of grammar rules for composition and their associated typing judgements, we are able to form an enhanced parser for natural language which provides a mathematical representation of the meaning of sentences, as proposed by Heim and Kratzer (1998)

1 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.

1.1 Typing Category

1.1.1 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 $\star = \text{Obj}(\bar{\mathcal{C}}) / \mathcal{F}(\mathcal{L})$. Since $\mathcal{F}(\mathcal{L})$ does not induce an equivalence relation on $\text{Obj}(\bar{\mathcal{C}})$ 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 Melliès and Zeilberger (2015)). We also define \star_0 to be the set obtained when considering types which have not yet been *affected*, that is $\text{Obj}(\mathcal{C})$. 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 $\perp \rightarrow \tau$ which we will denote directly by $\tau \in \star_0$.

1.1.2 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:

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

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 $\star \rightarrow \star$ which is a subset of \star . This explains how functors can act on functions.

¹We can even close it if we want to.

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:

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

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

1.1.3 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. If we have an adjunction $L \dashv R$, we have natural transformations for $\text{Id}_C \Rightarrow L \circ R$ and $R \circ L \Rightarrow \text{Id}_C$.
2. Though I don't have an example in the english language yet, there could be words which act as functor modifiers 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 $F \xRightarrow{\theta} G$ then for all arrows $f : \tau_1 \rightarrow \tau_2$ we have:

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

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 Bauer and Pretnar (2014) or Plotkin and Pretnar (2013).

1.1.4 Natural Transformations for Higher-Order Constructs

We might want to add plurals, superlatives, tenses, aspects and other similar constructs which act as function modifiers. 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 $\tau_1 = \perp$). It is associated with the following function (which is a rewriting of the natural transformation rule):

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

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

$\forall F \in \mathcal{F}(\mathcal{L}),$	$\frac{\Gamma \vdash x : \tau \quad \Gamma \vdash F : S \subseteq \star \quad \overbrace{\tau \in S}^{\exists \tau' \in S, \tau \preceq \tau'}}{\Gamma \vdash Fx : F\tau \preceq \tau}$	Cons
	$\frac{\Gamma \vdash x : \tau \quad \tau \in \star_0}{\Gamma \vdash Fx : F\tau \notin \star_0}$	FT_0
	$\frac{\Gamma \vdash x : F\tau_1 \quad \Gamma \vdash \varphi : \tau_1 \rightarrow \tau_2}{\Gamma \vdash \varphi x : F\tau_2}$	fmap
	$\frac{\Gamma \vdash x : \tau_1 \quad \Gamma \vdash \varphi : \tau_1 \rightarrow \tau_2}{\Gamma \vdash \varphi x : \tau_2}$	App
	$\frac{\Gamma \vdash x : A\tau_1 \quad \Gamma \vdash \varphi : A(\tau_1 \rightarrow \tau_2)}{\Gamma \vdash \varphi x : A\tau_2}$	$<*\rangle$
	$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash x : A\tau}$	pure/return
	$\frac{\Gamma \vdash x : MM\tau}{\Gamma \vdash x : M\tau}$	$\gg=$
$\forall F \xRightarrow{\theta} G,$	$\frac{\Gamma \vdash x : F\tau \quad \Gamma \vdash G : S' \subseteq \star \quad \tau \in S'}{\Gamma \vdash x : G\tau}$	nat

Table 1: Typing and Subtyping Judgements

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 reconcile 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:

$$\mathbf{future}(\mathbf{be})(\mathbf{arg}_1, \mathbf{arg}_2) = \mathbf{future}(\mathbf{be})(\mathbf{arg}_2)(\mathbf{future}(\mathbf{arg}_1)) = \mathbf{future}(\mathbf{be})(\mathbf{future}(\mathbf{arg}_2))(\mathbf{future}(\mathbf{arg}_1))$$

This is not a definitive rule as we could want to stop at a step in the derivation, depending on our understanding of the notion of future in the language.

1.2 Typing Judgements

To complete this section, Table 1 gives 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 inferring it at runtime). We could also add symmetry induced rules for application.

Furthermore, note that the App rule is the **fmap** rule for the identity functor and that the **pure/return** and $\gg=$ is the **nat** rule for the monad unit (or applicative unit) and multiplication.

S \mathbf{t}	ADJ(P) Either:
CN(P) $\mathbf{e} \rightarrow \mathbf{t}$	$(\mathbf{e} \rightarrow \mathbf{t})$ <i>carnivorous, happy</i>
ProperN \mathbf{e}	$(\mathbf{e} \rightarrow \mathbf{t}) \rightarrow (\mathbf{e} \rightarrow \mathbf{t})$ <i>skillful</i>
NP Either:	REL $\mathbf{e} \rightarrow \mathbf{t}$
\mathbf{e} <i>John, a cat</i>	VP, IV(P) $\mathbf{e} \rightarrow \mathbf{t}$
$(\mathbf{e} \rightarrow \mathbf{t}) \rightarrow \mathbf{t}$ <i>the cat, every man</i>	TV(P) $\Upsilon(\mathbf{NP}) \rightarrow \Upsilon(\mathbf{VP})$
DET $\Upsilon(\mathbf{CN}) \rightarrow \Upsilon(\mathbf{NP})$	<i>is</i> $(\mathbf{e} \rightarrow \mathbf{t}) \rightarrow \mathbf{e} \rightarrow \mathbf{t}$

Table 2: Usual Typings for some Syntactic Categories

2 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.

2.1 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 definitions³. Those are based on Partee. Here Υ is the operator which retrieves the type from a certain syntactic category.

2.2 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: $\Gamma \vdash \lambda x. \mathbf{planet}x : \mathbf{e} \rightarrow \mathbf{t}$. 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**. For 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.

Let us explain a few of those functors: **G** designates reading from a certain environment of type **r** while **W** encodes the possibility of logging a message of type **t** along with the expression. The functor **M** describes the possibility for a computation to fail, for example when retrieving a value that does not exist (see **the**). The **S** functor represents the space of possibilities that arises from a non-deterministic computation (see **which**).

We can then define an adjunction between **G** and **W** using our definitions:

$$\varphi : \left(\alpha \rightarrow \mathbf{G}\beta \right) \rightarrow \mathbf{W}\alpha \rightarrow \beta \quad \text{and} \quad \psi : \left(\mathbf{W}\alpha \rightarrow \beta \right) \rightarrow \alpha \rightarrow \mathbf{G}\beta$$

$$\lambda k. \lambda (a, g). kag \quad \lambda c \lambda a \lambda g. c(a, g)$$

where $\varphi \circ \psi = \text{id}$ and $\psi \circ \varphi = \text{id}$ on the properly defined sets. Now it is easy to see that φ and ψ do define a natural transformation and thus our adjunction $\mathbf{W} \dashv \mathbf{G}$ is well-defined, and that its unit η is $\psi \circ \text{id}$ and its co-unit ε is $\varphi \circ \text{id}$.

As a reminder, this means our language canonically defines a monad $\mathbf{G} \circ \mathbf{W}$. Moreover, the co-unit of the adjunction provides a canonical way for us to deconstruct entirely the comonad $\mathbf{W} \circ \mathbf{G}$, that is we have a natural transformation from $\mathbf{W} \circ \mathbf{G} \Rightarrow \text{Id}$.

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

Expression	Type	λ -Term
planet	$\mathbf{e} \rightarrow \mathbf{t}$ Generalizes to common nouns	$\lambda x.\mathbf{planet} \ x$
carnivorous	$(\mathbf{e} \rightarrow \mathbf{t})$ Generalizes to predicative adjectives	$\lambda x.\mathbf{carnivorous} \ x$
skillful	$(\mathbf{e} \rightarrow \mathbf{t}) \rightarrow (\mathbf{e} \rightarrow \mathbf{t})$ Generalizes to predicate modifier adjectives	$\lambda p.\lambda x.px \wedge \mathbf{skillful} \ x$
Jupiter	\mathbf{e} Generalizes to proper nouns	$\mathbf{j} \in \text{Var}$
sleep	$\mathbf{e} \rightarrow \mathbf{t}$ Generalizes to intransitive verbs	$\lambda x.\mathbf{sleep} \ x$
chase	$\mathbf{e} \rightarrow \mathbf{e} \rightarrow \mathbf{t}$ Generalizes to transitive verbs	$\lambda x.\lambda y.\mathbf{chase} \ (x, y)$
be	$(\mathbf{e} \rightarrow \mathbf{t}) \rightarrow \mathbf{e} \rightarrow \mathbf{t}$	$\lambda p.\lambda x.px$
she	$\mathbf{e} \rightarrow \mathbf{e}$	$\lambda x.x$
it	$(\perp \rightarrow \mathbf{Ge})$	$\lambda g.g_0$
which	$(\mathbf{e} \rightarrow \mathbf{t}) \rightarrow \mathbf{Se}$	$\lambda p.\{x \mid px\}$
the	$(\mathbf{e} \rightarrow \mathbf{t}) \rightarrow \mathbf{Me}$	$\lambda p.x \text{ if } p = \{x\} \text{ else } \#$
a	$(\mathbf{e} \rightarrow \mathbf{t}) \rightarrow \mathbf{De}$	$\lambda p.\lambda s.\{\langle x, x + s \rangle \mid px\}$
no	$(\mathbf{e} \rightarrow \mathbf{t}) \rightarrow \mathbf{Ce}$	$\lambda p.\lambda c.\neg \exists x.px \wedge c \ x$
every	$(\mathbf{e} \rightarrow \mathbf{t}) \rightarrow \mathbf{Ce}$	$\lambda p.\lambda c.\forall x.px \Rightarrow c \ x$

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

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

Table 4: Denotations for the functors used

CN(P)	$\Gamma \vdash p : (\mathbf{e} \rightarrow \mathbf{t})$	$\Pi(p) = \lambda x. (px \wedge x \geq 1)$
ADJ(P)	$\Gamma \vdash p : (\mathbf{e} \rightarrow \mathbf{t})$	$\Pi(p) = \lambda x. (px \wedge x \geq 1)$
	$\Gamma \vdash p : (\mathbf{e} \rightarrow \mathbf{t}) \rightarrow (\mathbf{e} \rightarrow \mathbf{t})$	$\Pi(p) = \lambda \nu. \lambda x. (p(\nu)(x) \wedge x \geq 1)$
NP	$\Gamma \vdash p : \mathbf{e}$	$\Pi(p) = p$
	$\Gamma \vdash p : (\mathbf{e} \rightarrow \mathbf{t}) \rightarrow \mathbf{t}$	$\Pi(p) = \lambda \nu. p(\Pi \nu)$
IV(P)/VP	$\Gamma \vdash p : \mathbf{e} \rightarrow \mathbf{t}$	$\Pi(p) = \lambda o. (po \wedge x \geq 1)$
TV(P)	$\Gamma \vdash p : \mathbf{e} \rightarrow \mathbf{e} \rightarrow \mathbf{t}$	$\Pi(p) = \lambda s. \lambda o. (p(s)(o) \wedge s \geq 1)$
REL(P)	$\Gamma \vdash p : \mathbf{e} \rightarrow \mathbf{t}$	$\Pi(p) = \lambda x. (px \wedge x \geq 1)$
DET	$\Gamma \vdash p : (\mathbf{e} \rightarrow \mathbf{t}) \rightarrow ((\mathbf{e} \rightarrow \mathbf{t}) \rightarrow \mathbf{t})$	$\Pi(p) = \lambda \nu. p(\Pi \nu)$
	$\Gamma \vdash p : (\mathbf{e} \rightarrow \mathbf{t}) \rightarrow \mathbf{e}$	$\Pi(p) = \lambda \nu. p(\Pi \nu)$

Table 5: Definition for the Π Plural Functor

2.3 Modality, Plurality, Aspect, Tense

We will now explain ways to formalise higher-order concepts as described in Section 1.1.4. First, we start by providing an example and then explaining how to modify it.

2.3.1 An Example: The *Plural*

Let us consider in this example the plural concept. Here we do not focus on whether our denotation of the plural is semantically correct, but simply on whether our modelisation makes sense. As such, we give ourselves a way to measure if a certain x is plural or not, which we will denote by $|x| \geq 1$. It is important to note that if τ is a type, and $\Gamma \vdash x : \tau$ then we should have $|\Pi x| \geq 1 = \top$. We then need to define the functor Π which models the plural. We do not need to define it on types in any way other than $\Pi(\tau) = \Pi\tau$. We only need to look at the transformation on arrows⁴. This one depends on the *syntactic*⁵ type of the arrow:

See that our functor only acts on the which return a boolean by adding a plurality condition, and is simply passed down else. Note however there is an issue with the **NP** category: in the case where a **NP** is constructed from a determiner and a common noun (phrase) the plural is not passed down. This comes from the fact that in this case, either the determiner or the common noun (phrase) will be marked, and by functoriality the plural will go up the tree.

Now clearly our functor verifies the identity and composition laws and works as theorised in Section 1.1.4. To understand a bit more why this works as described in our natural transformation rules, consider $\Pi(\mathbf{which})$. The result will be of type $\Pi(\mathbf{S}(e)) = (\Pi \circ \mathbf{S})(e)$. However, when looking at our transformation rule (and our functorial rule) we see that the result will actually be of type $\mathbf{S}(\Pi e)$ which is exactly the expected type when considering the phrase **which** p . The natural transformation θ we set is easily inferable:

$$\begin{aligned} \theta_A : (\Pi \circ \mathbf{S}) A &\xrightarrow{\theta_A} (\mathbf{S} \circ \Pi) A \\ \Pi(\{x\}) &\longmapsto \{\Pi(x)\} \end{aligned}$$

The naturality follows from the definition of **fmap** for the \mathbf{S} functor. We can easily define natural transformations for the other functors in a similar way: $\Pi \circ \mathbf{F} \xrightarrow{\theta} \mathbf{F} \circ \Pi$ defined by $\theta_\tau = \mathbf{fmap}_{\mathbf{F}}(\Pi)$.

⁴**fmap**, basically.

⁵Actually it depends on the word considered, but since the denotations (when considered effect-less) provided in Table 3 are more or less related to the syntactic category of the word, our approximation suffices.

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 $\perp \rightarrow \tau$ or $\tau \in \text{Obj}(\mathcal{C}_0)$. 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 $(\perp \rightarrow \tau, \square)$, 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 functions⁶.

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 $\star\tau_1 \rightarrow \star\tau_2$.

B Bibliography

- Andrej Bauer and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science*, Volume 10, Issue 4.
- Dylan Bumford. Effectful composition in natural language semantics - Introducing functors.
- Dylan Bumford and Simon Charlow. Effect-driven interpretation.
- Irene Heim and Angelika Kratzer. 1998. *Semantics in Generative Grammar*. Number 13 in Blackwell Textbooks in Linguistics. Blackwell, Malden, MA.
- Paul-André Melliès and Noam Zeilberger. 2015. 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. ACM.
- E. Moggi. 1989. Computational lambda-calculus and monads. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, CA, USA. IEEE Comput. Soc. Press.
- B Partee. Lecture 2. Lambda abstraction, NP semantics, and a Fragment of English. *Formal Semantics*.
- Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science*, Volume 9, Issue 4.

⁶It is!