A Taste of Categorical Semantics

Marco Paviotti

December 19, 2025

Abstract

A short race towards the semantics of the λ-calculus.

Contents

1 Introduction

When learning an abstract mathematical concept, it is helpful to have a concrete notion of the subject being studied; without this, the abstraction may lack meaningful context. There are probably two approaches to learning category theory in computer science: through the lens of mathematics or through that of functional programming. While the mathematical perspective is arguably the most rigorous, many computer scientists may find the functional programming perspective more accessible. Indeed, some authors have already chosen both the former [1, 3] and the latter [5].

In these notes, we take a different approach – one which lies between mathematics and programming languages. This approach aligns with the mathematical treatment of programming languages known as *denotational semantics*. The idealized concept introduced by Dana Scott [7] in 1969 is that a *type* should be viewed as a *set*, and a *program* as a *function* between sets. However, as programming languages incorporate more features, maintaining this simplistic view becomes increasingly challenging. By employing category theory, we generalize this perspective: a type corresponds to an *object*, and a program corresponds to an *arrow*. With this idea in mind, we will model the simply typed λ-calculus and λ-calculi with computational effects.

For the interested reader who wants to dive deeper in these subjects, there is a plethora of very well-written books about category for a more in-depth introduction on the subject [1, 3].

These notes however, focus on the applications of category theory to denotational semantics of programming languages, type theory and, hence, logic. Traditionally denotational semantics has been presented through the lenses of languages with recursion, thus using CPO semantics [9, 2, 8]. For the purpose of this presentation this is in a way just a particular case of the computational lambda calculus where recursion is a computational effect. Our aim is to maintain the view that category theory is a unifying theory covering a wide variety of subjects; from logic to type theory, hence programming, algebra and topology and it is therefore a subject worth studying for the reader's own good.

To give a taste for why this is true let us look at three most common axioms which we can find in logic, type theory and algebra. In particular, reflexivity and transitivity.

Reflexivity states that every proposition A entails itself, written $A \vdash A$ in logic. In type theory, this corresponds to the *variable rule*, which states that if a variable x has type A, written $x : A$, then we can type check the program $x : A$. This is written using the judgment $x : A \vdash x : A$.

Similarly, transitivity ensures that proofs of propositions can be chained: if $A \vdash B$ and $B \vdash C$ then $A \vdash C$. This is similar to what happens in type theory. Whenever a

program t has type A in a context Γ we write $\Gamma \vdash t : A$. Then if $x : A \vdash t : B$ is a program of type B with an open variable of type A and $y : B \vdash t' : C$ is a program of type C open in $y : B$ then substituting t for y in t',written $t'[t/y]$, yields a program of type C open in $x : A$.

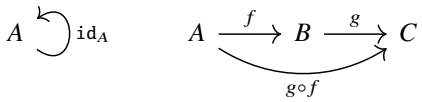
In algebra, a *preorder* has exactly the same properties as for all A in a preorder we have $A \leq A$ and for all A, B, C, if $A \leq B$ and $B \leq C$ then obviously $A \leq C$. The table below summarises the concepts we explained so far:

Logic	Type Theory	Algebra
$\mathbf{A} \vdash \mathbf{A}$	$x : \mathbf{A} \vdash x : \mathbf{A}$	$\mathbf{A} \leq \mathbf{A}$
$\dfrac{\mathbf{A} \vdash \mathbf{B} \quad \mathbf{B} \vdash \mathbf{C}}{\mathbf{A} \vdash \mathbf{C}}$	$\dfrac{x : \mathbf{A} \vdash t : \mathbf{B} \quad y : \mathbf{B} \vdash t' : \mathbf{C}}{x : \mathbf{A} \vdash \{t/y\}t' : \mathbf{C}}$	$\dfrac{\mathbf{A} \leq \mathbf{B} \quad \mathbf{B} \leq \mathbf{C}}{\mathbf{A} \leq \mathbf{C}}$

In the table above, it is evident that if in type theory we remove the programs and leave the types we end up with the same rules in logic and that if in logic we interpret judgments ($\vdash$) as $\leq$ then we obtain the same axioms of the preorder.

It should now be evident that three seemingly disconnected areas of mathematics – logic, computer science, and algebra – are deeply interconnected. This connection has been termed *computational trinitarianism*, the idea that logic, type theory, and algebra are three perspectives on the same underlying objects. This perspective reinforces the thesis that computability is an inherent concept in the foundations of mathematics.

In 1945, Saunders MacLane and Samuel Eilenberg, while working on algebraic topology, observed recurring patterns in algebra that could be generalized. He developed an abstract axiomatization encompassing many aspects of algebra, which led to the formulation of the *axioms of a category* [3]. A category consists of objects and morphisms (arrows) between them, governed by two fundamental axioms. The first, *identity*, reflects reflexivity by requiring an identity morphism $\mathtt{id}_A : A \to A$ for every object. The second, *composition*, embodies transitivity by allowing morphisms $f : A \to B$ and $g : B \to C$ to compose into $g \circ f : A \to C$.



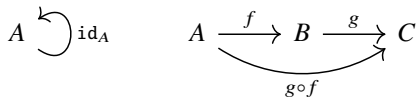
The reader is encouraged to keep these analogies in mind throughout this article, as they provide intuitive insights into the abstract categorical concepts we will explore.

An important thing to keep in mind is that all definitions here assume set theory as a foundation and in particular ZFC. The reader who is not familiar with set theory and the Russel's paradox can find a summary of these concepts in the Appendix A.

2 Categories

As explained above, the intuition the reader should have is that when talking about well-typed programming languages, types should be regarded as *objects* and programs should be regarded as *arrows*. This sort of motivates the definition of a category.

Definition 2.1 (Category)**.** *A* category *C is a collection of objects $A, B, C \ldots$ denoted by Obj(C) and a set of arrows $f, g, h \ldots$ denoted by Arr(C). Additionally, for each object A there exists an identity arrow $\mathtt{id}_A : A \to A$ such that and for arrows $f : A \to B$ and $g : B \to C$ we there exists an arrow $g \circ f : A \to C$. We picture these as mentioned in the introduction:*



Additionally, these arrows obey the identity and associativity laws:

$$\mathtt{id}_A \circ f = f \circ \mathtt{id}_A = f$$
$$f \circ (g \circ h) = (f \circ g) \circ h$$

Examples of categories are the category of sets, denoted by **Set**, where objects are sets and arrows are functions, the category of sets and relations **Rel**, the category of partial order sets **Pos** and monotone functions, the category of groups **Grp** of groups and group homomorphisms, the category of topological spaces **Top** of topological spaces and continuous functions between them.

To avoid clutter, we simply write $X \in C$ for an object in a category C and $f \in C$ for a morphism in a category C. For objects $X, Y \in C$ we write $C(X, Y)$ for the collection of morphisms from X to Y which we call the homset.

The fact that a category is defined in terms of collections objects and morphisms is to avoid paradoxes such as the one in Section A.4. For example, the category **Set** is too big for the objects to form a set, since the set of sets does not exists. When the collection of objects is a set we say the category is *small*. Similarly when the homset is a set we say the category is *locally small*.

2.1 Isomorphisms

Isomorphism is a fundamental concept that captures the idea of two objects being "essentially the same". An isomorphism between two objects indicates that they are structurally identical, even if they might appear different externally. While the concept of "essentially the same" is central to isomorphism, it is not always straightforward to understand what this means in different settings. An isomorphism in **Set** is a pair of functions which are inverses to each other. This corresponds to saying that two sets are isomorphism if they have the same cardinality, which is equivalent to saying that there exists a function $f : A \to B$ such that is surjective and injective.

However, consider the the category **Pos** of partial order sets and order preserving functions. The following are two posets which are not isomorphic:

since in the right-hand side poset the $\perp$ element is not ordered with 1. Despite the fact that these two posets are in bijection they are not isomorphic because any bijection would be able to preserve the order $\perp \leq 1$ from the left to the right-hand side poset (while still being a bijection).

Category theory abstracts the notion of isomorphism by stating that two objects are isomorphic if and only if there exists a pair of morphisms which are inverse to each other

Definition 2.2 (Isomorphism)**.** *Two given objects A and B are isomorphic, written $A \cong B$ iff there exists an arrow $f : A \to B$ that has an inverse $g : B \to A$ such that*

$$g \circ f = id_A \qquad f \circ g = id_B$$

This definition depends of course on the definition of morphism, in particular, an isomorphism is defined by what the arrows look like and by what we can observe through them rather than what are the objects themselves.

3 Models of the Simply Typed λ-Calculus

3.1 Initial and terminal objects

Consider the false proposition in logic. When we assume something absurd we can derive anything, hence a false proposition entails any formula A. In programming this is expressed in terms of empty data types which are denoted sometimes with 0. Having a piece of data of type 0, say $x : 0$, is absurd since the empty type contains nothing, thus performing case analysis on this data will be vacuous as state in the axiom below

$$\frac{}{x : 0 \vdash \text{case } x \text{ of } \{\} : A}$$

The axiom above may look strange at first. The variable x is case analysed on but there is no term inside the case operator. But this makes sense because case analysis on a data structure means providing a program to continue with for each constructor in the data structure, however there is no constructor in the empty type so there is nothing further we need to provide.

In category theory this is expressed in terms of initial objects. The initial object is an object denoted by 0 such that for every other object A there exist a unique arrow between 0 and A. We draw a dashed arrow as follows to indicate the arrow is unique:

$$0 \overset{!_A}{\dashrightarrow} A$$

In other words, there is a unique proof which takes a proof of the false statement and returns a proof of any statement A. This can also be interpreted in programming as the program from the empty type into any type. Intuitively, the only way to produce something of type A is to case analyse on the empty type, but because this type does not contain anything the program has nothing further to compute and the case is vacuous.

An equivalent formulation is by viewing what happens in the homsets. In particular,

$$C(0, A) \cong \mathbf{1}(*, *)$$

natural in A. This can be proven using the uniqueness property which is more formally stated as follows:

$$f = !_A \qquad \text{(Uniqueness Property)} \tag{1}$$

for all $f : 0 \to A$. This implies a reflection and fusion law:

$$id_0 = !_0 \qquad \text{(Reflection Law)} \tag{2}$$

$$k \circ !_A = !_B \qquad \text{(Fusion Law)} \tag{3}$$

Similarly, the *terminal* object 1 represent the true statement or the type with only one element in it, the *unit type*. In programming terms, given any input of type A there exists exactly one program producing an element of the unit type, that is the program which discards the input and returns the single element in the unit type. Categorically, and it is such that for every object A there is a unique arrow into the terminal object:

$$A \overset{!_A}{\dashrightarrow} 1$$

It is important to know that such objects in category theory are unique only up-to isomorphism. Similarly to the initial object we can view this property also from the point of view of representable homsets:

$$\mathbf{1}(*, *) \cong C(A, 1)$$

natural in A. Formally, this can be derived by the uniqueness property of the terminal object can be stated as follows::

$$f = !_A \qquad \text{(Uniqueness Property)} \tag{4}$$

for all $f : A \to 1$. This implies a reflection and fusion law:

$$id_1 = !_1 \qquad \text{(Reflection Law)} \tag{5}$$

$$!_B \circ k = !_A \qquad \text{(Fusion Law)} \tag{6}$$

In **Set**, the empty set $\emptyset$ is the initial object since there is a unique function from the empty set to any other set A, that is the empty function or empty relation. Similarly, the terminal object is any set containing exactly one element, the singleton set. Consider the set $\{*\}$. For any other given set A there is a unique function into $\{*\}$ which is the constant function mapping every element $x \in A$ into $\{*\}$. Notice that there are many terminal objects in **Set**, but they are all isomorphic in that they all contain only one element. Also **Set** is a special category of sorts, in fact, it enjoys the property that the set of morphisms from 1 to any set A is isomorphic to A itself since any function $1 \xrightarrow{x} A$ can map into exactly one element in A.

3.2 Products, CoProducts and Exponentials

In this section we are going to take a look at the categorical semantics of the simply typed λ-calculus. This is a λ-calculus with natural numbers, pairs and function types therefore it is only natural that to model these we need their logical and algebraic counterparts.

3.2.1 Products

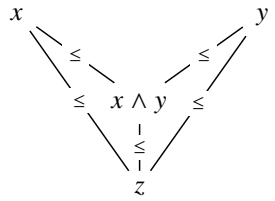
In logic, the *and* operator is introduced by stating that if Γ is a set of true propositions which entails A and this context Γ entails also B then of course $\Gamma \vdash B$. This is called the introduction rule of the product:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

The product has also two elimination rules given by:

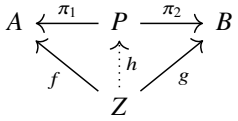
$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

In algebra, a set which possess this structures is called a *lower semilattice*. That is a set X such that for every $a, b \in X$, there exists an element $a \wedge b$ called the *meet* or the *greatest lower bound* of a and b which has the property that for every other lower bound z, that is $z \leq a$ and $z \leq b$ we have that $z \leq a \wedge b$.



We proceed now to generalise the concept of greatest lower bound to the categorical notion of product $A \times B$.

Definition 3.1 (Product)**.** *For two objects A and B the* product *is an object P equipped with two arrows $\pi_1 : P \to A$ and $\pi_2 : P \to B$ such that for any object Z with arrows $f : Z \to A$ and $g : Z \to B$ there exists a unique morphism $h : Z \to P$ making the following diagram commute:*



We write $A \times B$ for the product of two objects A and B and $\langle f, g \rangle$ for h.

From the definition of product we can derive the following isomorphism of arrows

$$C \times C((Z, Z), (A, B)) \cong C(Z, A \times B)$$

natural in $Z, A, B : C$. Intuitively, if we have two arrows $f : Z \to A$ and $g : Z \to B$ we can form the arrow $Z \to A \times B$ and viceversa from an arrow into the product we can obtain two arrows into A and B by post-composing with the respective projections.

The isomorphism is proven by using the uniqueness property which can be more formally written as follows:

$$\pi_1 \circ g = x \text{ and } \pi_2 \circ g = y \iff \langle x, y \rangle = g \tag{7}$$

for all $g : Z \to A \times B$. This implies the following properties:

$$\pi_1 \circ \langle x, y \rangle = x \text{ and } \pi_2 \circ \langle x, y \rangle = y \qquad \beta\text{-laws} \tag{8}$$

$$\langle \pi_1 \circ h, \pi_2 \circ h \rangle = h \qquad \eta\text{-law} \tag{9}$$

$$\langle \pi_1, \pi_2 \rangle = id_{A \times B} \qquad \text{Reflection Law} \tag{10}$$

$$\langle x, y \rangle \circ h = \langle x \circ h, y \circ h \rangle \qquad \text{Fusion Law} \tag{11}$$

$$(f \times g) \circ \langle x, y \rangle = \langle f \circ x, g \circ y \rangle \qquad \text{Functor Fusion Law} \tag{12}$$

The definition of product corresponds to the notion of product in **Set**

$$A \times B \stackrel{\Delta}{=} \{\langle a, b \rangle \mid a \in A \text{ and } b \in B\}$$

where the projections π_1 and π_2 are simply the functions discarding one component of the pair $\pi_1(a, b) =$ and $\pi_2(a, b) = b$ and for every element $1 \xrightarrow{a} A$ and $1 \xrightarrow{b} B$ the pairing is defined by $(a, b) \mapsto \langle a, b \rangle$ Note that the existence condition of the pairing function enforces that every element of A and B are in the product (no noise) and the uniqueness condition ensures that there is are no more elements in $A \times B$ than the ones that are coming from A and B (no junk).

Notice that we could have defined the product in many other ways, but as long as the categorical definition is satisfied, all these definitions are isomorphic. The reader should convince themselves that is true by proving the following proposition:

Proposition 3.1 (Products are unique up to isomorphism)**.** *Let C be a category with products. Then for all objects A and B, if P and Q are both products of A and B then $P \cong Q$.*

Another isomorphic definition of product in **Set** could be the following one:

$$A \times B = \{(b, a) \mid a \in A \text{ and } b \in B\}$$

or this one

$$A \times B = \{(b, a, a) \mid a \in A \text{ and } b \in B\}$$

3.2.2 Coproducts

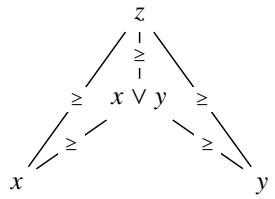
In logic, the *or* operator is introduced by stating that if Γ is a set of true propositions that entails A, or if Γ entails B, then $\Gamma \vdash A \vee B$. This is called the introduction rule of the coproduct:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

The coproduct has one elimination rule given by:

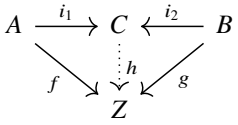
$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}$$

In algebra, a set with this structure is called an *upper semilattice*. That is, a set X such that for every $a, b \in X$, there exists an element $a \vee b$, called the *join* or the *least upper bound* of a and b, with the property that for every other upper bound z (i.e., $z \geq a$ and $z \geq b$), we have $z \geq a \vee b$. This can be visualized as follows:



We now generalize the concept of least upper bound to the categorical notion of a coproduct $A + B$.

Definition 3.2 (Coproduct)**.** *For two objects A and B, the* coproduct *is an object C equipped with two arrows $i_1 : A \to C$ and $i_2 : B \to C$ called* injections *such that for any object Z with arrows $f : A \to Z$ and $g : B \to Z$, there exists a unique morphism $h : C \to Z$ making the following diagram commute:*



We write $A + B$ for the coproduct of two objects A and B and $[f, g]$ for h.

It should be clear by now that we can view the properties of this objects under the lenses of the homsets as well:

$$C(A + B, Z) \cong C \times C((A, B), (Z, Z))$$

This is derived by the unique mapping property of the coproduct:

$$f = [g_1, g_2] \iff f \circ \texttt{inl} = g_1 \text{ and } f \circ \texttt{inr} = g_2 \tag{13}$$

This yields the following properties:

$$[g_1, g_2] \circ \texttt{inl} = g_1 \text{ and } [g_1, g_2] \circ \texttt{inr} = g_2 \qquad \beta\text{-law} \tag{14}$$

$$h = [h \circ \texttt{inl}, h \circ \texttt{inr}] \qquad \eta\text{-law} \tag{15}$$

$$[\texttt{inl}, \texttt{inr}] = id_{A+B} \qquad \text{Reflection Law} \tag{16}$$

$$f \circ [g_1, g_2] = [f \circ g_1, f \circ g_2] \qquad \text{Fusion Law} \tag{17}$$

$$[g_1, g_2] \circ h_1 + h_2 = [g_1 \circ h_1, g_2 \circ h_2] \qquad \text{Functor Fusion Law} \tag{18}$$

This definition corresponds to the notion of a disjoint union in **Set**:

$$A + B \stackrel{\Delta}{=} \{(a, 1) \mid a \in A\} \cup \{(b, 2) \mid b \in B\},$$

where the injections i_1 and i_2 are defined as:

$$i_1(a) = (a, 1) \quad \text{and} \quad i_2(b) = (b, 2).$$

For any element $f : A \to Z$ and $g : B \to Z$, the unique morphism $[f, g]$ is defined by:

$$[f, g](x) = \begin{cases} f(a) & \text{if } x = (a, 1), \\ g(b) & \text{if } x = (b, 2). \end{cases}$$

The existence condition ensures that every element of A and B is included in the coproduct (no omission), while the uniqueness condition ensures that there are no additional elements in $A + B$ (no duplication).

Notice that we could define the coproduct in many ways, but as long as the categorical definition is satisfied, all these definitions are isomorphic. The reader should verify this by proving the following proposition:

Proposition 3.2 (Coproducts are unique up to isomorphism)**.** *Let C be a category with coproducts. Then for all objects A and B, if C and D are both coproducts of A and B, then $C \cong D$.*

3.2.3 Exponentials

In logic, the introduction rule of the implication is typically written as follows:

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

where $\subset$ is how traditionally implication has been written and is reminiscent of the fact that the information A provides is a subset of the information that B provides. The introduction rule means that if we can derive B from a context Γ and a proof of A then we can derive that given Γ we can prove A implies B. The elimination rule of the implication is called "modus ponens" rule is typically written as follows:

$$\frac{\Gamma \vdash A \subset B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

meaning that if Γ entails A implies B and we also have that Γ entails A then we can derive B.

The algebraic structure modelling this operator is called an *Heyting algebra*. An Heyting algebra consists of a set X with finite meets ($\wedge$) and exponents ($\Rightarrow$) such that the following universal property holds

$$z \wedge x \leq y \iff z \leq x \Rightarrow y$$

for all x, y, z.

This time the categorical definition will not match exactly the abstract definition of the Heyting algebra. However we can show we can reduce the following definition to the one above for preorders.

Definition 3.3 (Exponentials)**.** *An* exponential *represents the* internal *function type of a language. The triangular diagram in (19) defines the universality property of an exponential. This is an object which we denote by B^A, for $A, B \in \mathcal{C}$ such that there exists an* evaluation map $\epsilon : B^A \times A \to B$ *which intuitively applies the input of type A to a function which takes A to B and such that for every map $Z \times A \xrightarrow{f} B$ there exists a unique map $Z \xrightarrow{\Lambda f} B^A$ such that the diagram in (19) commutes.*

$$\begin{array}{ccc} B^A & & B^A \times A \xrightarrow{\epsilon} B \\ \Lambda f \uparrow & & \Lambda f \times A \uparrow \quad \nearrow f \\ Z & & Z \times A \end{array} \tag{19}$$

Notice, here Λ can be seen as the currying operation taking maps of type $Z \times A \to B$ to maps to type $Z \to B^A$. Conversely, if we happen to have a map $Z \xrightarrow{g} B^A$ then we can easily construct a map $Z \times A \xrightarrow{\forall g} B$ by $\epsilon \circ \langle g \circ \pi_1, \pi_2 \rangle$. Thus it can be proven that there is a correspondence of arrows, in other words, a correspondence of homsets

$$\Lambda : \mathcal{C}(Z \times A, B) \cong \mathcal{C}(Z, B^A) : \forall$$

which corresponds to our definition of Heyting exponential and it is the categorical definition of currying and uncurrying.

This isomorphism is derived by the uniqueness property:

$$g = \Lambda f \iff \epsilon \circ g \times A = f \tag{20}$$

which yields the following additional properties:

$$\epsilon \circ \Lambda f \times A = f \qquad \beta\text{-law} \tag{21}$$

$$h = \Lambda(\epsilon \circ h \times A) \qquad \eta\text{-law} \tag{22}$$

$$\Lambda\epsilon = id_{B^A} \qquad \text{Reflection Law} \tag{23}$$

$$\Lambda f \circ g = \Lambda(f \circ g) \qquad \text{Fusion Law} \tag{24}$$

$$(g)^A \circ \Lambda f = \Lambda(g \circ f) \qquad \text{Functor Fusion Law} \tag{25}$$

3.3 Semantics of the Simply Typed λ-Calculus

In this section we look at a categorical semantics for the Simply-Typed λ-calculus (STLC) which was introduced by Alonzo Church in 1940.

3.3.1 Syntax

We first define the syntax of STLC by defining the set of types, the set of terms and the typing judgment relation. The set of types Types also inductively as follows

$$\begin{array}{rcll} A, B \in \text{Types} & ::= & \texttt{unit} & \text{(unit type)} \\ & | & A \times B & \text{(products)} \\ & | & A \to B & \text{(functions)} \end{array}$$

while the set of λ-terms Terms is inductively defined as follows

$$\begin{array}{lcll} t \in \text{Terms} & ::= & x & \text{(terms variables)} \\ & | & (t_1, t_2) \mid \mathtt{prj}_1(t) \mid \mathtt{prj}_2(t) & \text{(products)} \\ & | & \lambda x.t \mid t_1 t_2 & \text{(functions)} \end{array}$$

Typing Judgment The typing judgement relation $\vdash$ is defined inductively as follows

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{}{\Gamma \vdash () : \mathsf{unit}}$$

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathtt{prj}_1(t) : A} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathtt{prj}_2(t) : B} \qquad \frac{\Gamma \vdash t_1 : A \quad \Gamma \vdash t_2 : B}{\Gamma \vdash \langle t_1, t_2 \rangle : A \times B}$$

$$\frac{\Gamma \vdash t_1 : A \to B \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$$

3.3.2 Semantics

The semantics of a programming language is given by the *semantic function* mapping the syntax into the *meaning* of the language. This function is traditionally denoted by $[\![\cdot]\!]$ and pronounced the *semantic brackets*. We use the semantic brackets for both the types, terms and the context interpretation. First we need to interpret the contexts. Since these are essentially finite lists we need a category with *finite products*. Thus define $[\![\Gamma]\!]$ by assuming that every context Γ is a finite list of typed variables $x_1 : A_1, \cdots, x_n : A_n$:

$$[\![x_1 : A_1, \cdots, x_n : A_n]\!] = [\![A_1]\!] \times \cdots \times [\![A_n]\!]$$

Or equivalently, by interpreting the context by induction on the list:

$$\begin{aligned} [\![.]\!] &= 1 \\ [\![\Gamma, x : A]\!] &= [\![\Gamma]\!] \times [\![A]\!] \end{aligned}$$

which is a more intuitive definition for readers familiar with functional programming.

The interpretation of types is then a function $[\![\,]\!] : \text{Types} \to \text{Obj}(C)$ from syntactic types into object of a category enforcing the intuition that in categorical semantics we think of types as objects:

$$\begin{aligned} [\![\mathsf{unit}]\!] &= 1 \\ [\![A \times B]\!] &= [\![A]\!] \times [\![B]\!] \\ [\![A \to B]\!] &= [\![B]\!]^{[\![A]\!]} \end{aligned}$$

And, finally, the function $[\![\cdot]\!] : \text{Terms} \to \text{Arr}(C)$ interprets a term as a morphism between two objects in the category C. However, to be more precise we only interpret well-typed terms, thus it would be more correct to say that a typing derivation $\Gamma \vdash t : A$ is interpreted as an arrow $[\![t]\!]$ of type $[\![\Gamma]\!] \to [\![A]\!]$, therefore abusing some notation the correct type of the interpretation would be something like:

$$[\![\Gamma \vdash t : A]\!] : [\![\Gamma]\!] \xrightarrow{[\![t]\!]} [\![A]\!]$$

For example, if $\Gamma, x : A \vdash t : B$ then the interpretation of t has type

$$[\![\Gamma, x : A \vdash t : B]\!] : [\![\Gamma]\!] \times [\![A]\!] \to [\![B]\!]$$

We now define this function by induction on the typing judgement:

$$\begin{aligned}
[\![\Gamma \vdash () : \mathtt{unit}]\!] &= ! \\
[\![\Gamma \vdash x : A]\!] &= \pi_x \\
[\![\Gamma \vdash \mathtt{prj}_1(t) : A]\!] &= \pi_1 \circ [\![t]\!] \\
[\![\Gamma \vdash \mathtt{prj}_2(t) : B]\!] &= \pi_2 \circ [\![t]\!] \\
[\![\Gamma \vdash (t_1, t_2) : A \times B]\!] &= \langle [\![t_1]\!], [\![t_2]\!] \rangle \\
[\![\Gamma \vdash \lambda x.M : A \to B]\!] &= \Lambda [\![M]\!] \\
[\![\Gamma \vdash MN : B]\!] &= \epsilon \circ \langle [\![M]\!], [\![N]\!] \rangle
\end{aligned}$$

where $! : [\![\Gamma]\!] \to 1$ is the unique map into the terminal object and $n : \Gamma \to \mathbb{N}$ is the map disregarding the context and returning the number denoted syntactically by $\underline{n}$. In **Set**, for example, there would be a constant map $\lambda\gamma.n$ for each n.

4 Functors

In this section we are going to expound the definition of functors. A functor $F : \mathcal{C} \to \mathcal{D}$ is essentially an arrow-preserving mapping between categories, mapping objects to object and arrows to arrows.

To give an intuition for what a functor is, consider first an *endofunctor* $\boxed{\cdot} : \mathcal{C} \to \mathcal{C}$ taking an object A into an object $\boxed{\mathrm{A}}$ in the same category. Imagine now that A is a type, then $\boxed{\mathrm{A}}$ is a parametrised type wrapping the elements of A around some structure. The central feature of a functor is that given a map $f : A \to B$ we can go inside the box and change the data inside using f. This is made more formal by saying that an endofunctor has always a map

$$\mathtt{fmap}\ f : \boxed{\mathrm{A}} \to \boxed{\mathrm{B}}$$

transforming the data inside a box while preserving the structure of the box itself. Examples of endofunctors include lists, trees, bags, exception types and even monads (see Section 5). Generalising, we can say the box is a *functor* F and the $\mathtt{fmap}\ f$, more commonly written $F(f)$, is the *functorial action* of F. Thus, it has two components:

- It takes each object X in one category and assigns it to an object $F(X)$ in another.
- It takes each morphism $f : X \to Y$ and assigns it to a morphism $F(f) : F(X) \to F(Y)$, keeping the relationships intact.

with the additional conditions that that it also also respects how morphisms behave, that is it respects identity morphisms and it respects composition. The formal definition follows:

Definition 4.1 (Functor)**.** *Let $\mathcal{C}$ and $\mathcal{D}$ be two categories. A* functor $F : \mathcal{C} \to \mathcal{D}$ *is a mapping between categories that associates objects in $\mathcal{C}$ to an object $F\mathcal{C} \in Obj(\mathcal{C})$ and that has additionally a* functorial action *associating arrows $f \in \hom(A, B)$ to arrows $F(f) \in \hom(FA, FB)$ which additionally preserves identity and composition:*

$$F(id_A) = id_{FA} \qquad F(g \circ f) = F(g) \circ F(f)$$

Another way of looking at functors is as order-preserving maps between categories. We covered briefly the concept of preorders as particular sets with an additional structure. In a preorder $(X, \leq)$, elements have a sense of order: if $a \leq b$, then a comes before b. A *structure-preserving* map between preorders is called a *monotone function*. This map respects the order of the set X in the following way:

$$\text{if } a \leq b \text{ then } f(a) \leq f(b)$$

Categories are a richer version of a preorder, where instead of just "$\leq$", we have morphisms (arrows) between objects. The `fmap` is the property witnessing the fact the every functor preserves arrows. In **Set** we can write:

$$\texttt{fmap} : (A \to B) \to FA \to FB$$

We now list some example of endofunctors in **Set**. A very popular one is the "maybe" functor

$$\texttt{Maybe}\ X = X + 1$$

The injections into the coproduct (+) give the operations $\texttt{success}(x)$ and `error`. In the former case is equivalent to write $\texttt{inl}(x)$, in the latter case this is equivalent to writing $\texttt{inr}(*)$. The functorial action is defined as $F(f) = f + 1$ which is a map of type $X + 1 \to Y + 1$ by applying f to the left part of the coproduct. This is formally defined using the unique mapping property of the coproduct (left as exercise).

Another very popular functor is list functor defined as the least solution to the following domain equation

$$\texttt{List}\ A \cong 1 + A \times \texttt{List}\ A$$

where $\phi : 1 + A \times \texttt{List}\ A \to \texttt{List}\ A$ is an isomorphism. We can define the empty list and the cons operator by

$$\texttt{[]} = \phi(\texttt{inl}(*)) \qquad x\texttt{:}xs = \phi(\texttt{inr}(\langle x, xs \rangle))$$

The functorial action is defined by $\texttt{List}\ f = 1 + f \times \texttt{List}\ A$, using the functorial action of the coproduct and the product, since both are functors themselves (prove it as an exercise).

Finally, the stream functor is defined as the greatest solution to the following equation

$$\texttt{Str}\ A \cong A \times \texttt{Str}\ A \tag{26}$$

The head and tail of a stream can be defined as

$$\texttt{head}(s) = \pi_1(\texttt{cons}^{-1}(s)) \qquad \texttt{tail}(s) = \pi_2(\texttt{cons}^{-1}(s))$$

where $\texttt{cons}^{-1}$ is the inverse of the cons operator $\texttt{cons} : A \times \texttt{Str}\ A \to \texttt{Str}\ A$ and π_1 and ϕ_2 are the first and second projections out of the product.

The functorial action is defined by $\texttt{Str}\ (f) = f \times \texttt{Str}\ A$ using the functorial action of the product.

We list some more examples of functors, the interested reader is invited to work out the details of these definitions.

Example 4.1. *Some popular functors include:*

- *the identity functor $FX = X$ with the obvious functional action*
- *the product functor (on one variable) $FX = A \times X$ with functorial action $F(f) = A \times f$*
- *the exponential $FX = X^A$ with the functorial action given by post-composition, that is $F(f) = \Lambda(f \circ \epsilon)$ where $\epsilon : X^A \times A \to Y$ is the evaluation map*
- *the* homset functor $FX = C(A, X)$. *This will be discussed more in depth later*

From the definition of functor it follows almost directly that every functor preserves isomorphisms:

$$A \cong B \Rightarrow FA \cong FB$$

the converse is not true unless the functor is surjective on morphisms, that is the functor is *full*. When the functorial action is injective the functor is called *faithful*.

Proposition 4.1. *A fully faithful functor F preserves and reflects isomorphisms:*

$$A \cong B \iff FA \cong FB$$

Exercise 4.1. *Prove Proposition 4.1*

4.1 Natural Transformations

A natural transformation is a *uniform mapping* between functors. What this means is that it transforms the structure of a functor into another functor without having any knowledge of the particular shape of the data inside it. For example, say that $\boxed{\text{A}}$ is a list of elements of type A and $\textcircled{\text{A}}$ is a tree of elements of A. A natural map ψ is a family of maps for each object A

$$\psi_A : \boxed{\text{A}} \to \textcircled{\text{A}}$$

which converts a list in to a tree, but without knowing anything about the object A. For example, suppose that you want to write an algorithm which converts a computation on X which may fail into the list of values X this computation can produce. This means we have to write a function from $\texttt{Maybe}\ X$ to $\texttt{List}\ X$ regardless of what X. The most natural way to do this is by mapping $\texttt{error}$ to the empty list $[]$, and $\texttt{success}(x)$ to the singleton list $[x]$.

This transformation is *agnostic* to the type of values. Since it holds for all type of values the conversion is not arbitrary but follows a deep categorical pattern. Thus, turning $\texttt{Maybe}$ into $\texttt{List}$ this way is a natural transformation.

Definition 4.2 (Natural Transformation)**.** *For two functors $F, G : C \to \mathcal{D}$, a* natural transformation *is a family of arrows $\phi_A : FA \to GA$ such that for every arrow $f : A \to B$ the following diagram commutes:*

$$\begin{array}{ccc} FA & \xrightarrow{\phi_A} & GA \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ FB & \xrightarrow{\phi_B} & GB \end{array}$$

A natural transformation $\psi : FA \to GB$ can be seen in **Set** as a function

$$\psi : \forall A.FA \to FB$$

were $\forall$ is for now an operator quantifying over all sets. In other words, natural transformations can be thought of as *polymorphic functions*.

4.2 Constructions on Categories

The *category of functors*, denoted as $[C, \mathcal{D}]$, is a category where the objects are functors $F : C \to \mathcal{D}$, where C and $\mathcal{D}$ are two categories. The morphisms in $[C, \mathcal{D}]$ are natural transformations between functors.

An isomorphism in $[C, \mathcal{D}]$ is a *natural isomorphism*, which means that for a natural transformation $\phi : F \to G$, each $\phi_X : F(X) \to G(X)$ is an isomorphism in $\mathcal{D}$. This implies that F and G are isomorphic functors in $[C, \mathcal{D}]$.

The *category of categories*, denoted **Cat**, is the category whose objects are *small categories* and whose morphisms are functors between them. A category consists of objects, morphisms (arrows), composition of morphisms that is associative, and identity morphisms. Since the collection of *all* categories is too large to form a category, **Cat** typically consists of *small categories*, meaning those whose collections of objects and morphisms form sets rather than proper classes. The morphisms in **Cat** are *functors*, which map objects and morphisms between categories while preserving composition and identity morphisms. Functors compose naturally: if $F : C \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{E}$ are functors, their composition $G \circ F : C \to \mathcal{E}$ is also a functor. The identity morphisms in **Cat** are identity functors, which map each object and morphism to itself. In addition to functors, there exist *natural transformations*, which provide structure-preserving ways to transition between functors. This makes **Cat** more than just a category – it is actually a *2-category*, where objects are categories, 1-morphisms are functors, and 2-morphisms are natural transformations between functors. This additional structure allows for deeper relationships between categories, making **Cat** fundamental in higher category theory.

Having the categories of categories allows us to talk about *isomorphisms of categories*. Following the definition of isomorphism these are the categories such that there exists a functor $F : C \to \mathcal{D}$ along with an inverse functor F^{-1}.

A *product category* refers to the *cartesian product* of two categories. Given two categories C and $\mathcal{D}$, their product category, denoted as $C \times \mathcal{D}$, is a category where objects are pairs (C, D), where C is an object from C and D is an object from $\mathcal{D}$.

Morphisms are pairs (f, g), where $f : C \to C'$ is a morphism in C, and $g : D \to D'$ is a morphism in $\mathcal{D}$. Composition of morphisms is defined component-wise as

$$(f', g') \circ (f, g) = (f' \circ f, g' \circ g).$$

Identity morphisms are given by $(\mathrm{id}_C, \mathrm{id}_D)$, where id_C and id_D are identity morphisms in C and $\mathcal{D}$, respectively.

An *opposite category* refers to a way of reversing the structure of a given category. If we have a category C, its opposite category, denoted C^{op}, is formed by reversing the direction of all morphisms while keeping the same objects.

For a category C, the category C^{op} is defined such that it has as objects the same objects as C and for every morphism $f : A \to B$ in C a morphism $f^{\mathsf{op}} : B \to A$. It follows straightforwardly that

$$f \in C(X, Y) \iff f^{\mathsf{op}} \in C^{\mathsf{op}}(Y, X)$$

The opposite category is a useful tool for exploring dualities, where a statement about C has a corresponding dual statement about C^{op}. We will go back to this topic when we introduce the concept of a functor in Section 4.

4.3 The Homset Functor

Given a (locally small) category C[1], the hom-set $C(A, B)$, for any objects A, B, induces two functors.

The first is the *covariant hom-functor* $C(A, -) : C \to \mathbf{Set}$, which assigns to each object B the set of morphisms $C(A, B)$. Given an arrow $f : B \to B'$ in C, the functor maps it to the function $C(A, f) : C(A, B) \to C(A, B')$, which acts by *post-composition*: each arrow $g : A \to B$ is sent to $f \circ g : A \to B'$.

The second is the *contravariant hom-functor* $C(-, B) : C^{\mathsf{op}} \to \mathbf{Set}$, which assigns to each object A the set $C(A, B)$. This functor is *contravariant* because it reverses arrows: a morphism $f : A \to A'$ in C is mapped to the function $C(f, B) : C(A', B) \to C(A, B)$, which acts by *pre-composition*: each arrow $g : A' \to B$ is sent to $g \circ f : A \to B$.

Functors which are *naturally isomorphic* to the functor $C(A, -)$ for some $A \in \mathrm{Obj}(()C)$ are called *representable functors*.

For example, the stream functor $\mathtt{Str} : \mathbf{Set} \to \mathbf{Set}$ is defined as the greatest solution to the domain equation

$$\mathtt{Str}A \cong A \times \mathtt{Str}A$$

This can be seen as the type of infinite lists over A. Let ι be the isomorphism, then the head of the stream is defined by post-composition of the first projection with ι and the tail by the second projection precomposed with ι.

In **Set**, the stream functor is naturally isomorphic to the functor $\mathrm{hom}_{\mathbf{Set}}(\mathbb{N}, -)$, i.e. the following isomorphism is natural in A

$$\mathtt{Str}\ A \cong \mathbf{Set}(\mathbb{N}, A) \tag{27}$$

[1] A category is **small** if its collection of objects forms a set, and it is **locally small** if, for any two objects A, B, the collection of arrows between them forms a set. However, we do not delve into size issues in these notes.

This is easy to see since we can take an infinite stream s and rename its elements such that each element has the natural number attached to it representing its position within the stream

$$a_1 a_2 \ldots a_n \ldots$$

This forms obviously a map $\mathbb{N} \to A$.

5 Monads

In Section 4 we viewed functors as boxes containing certain data. These boxes had the ability to transform the data inside them while preserving the outer structure of the box. In particular, for a give map $f : A \to B$ a functor can be viewed as a certain structure containing A, written $\boxed{\text{A}}$, along with a map which may be called $\texttt{fmap f} : \boxed{\text{A}} \to \boxed{\text{B}}$ transforming the data inside the box.

A *monad* is a functor, with additional capability of inserting data into the box via the "unit" map

$$\eta : A \to \boxed{\text{A}}$$

and to squash two layers into one via a "multiplication" map

$$\mu : \boxed{\boxed{\text{A}}} \to \boxed{\text{A}}$$

Importantly, these two maps are *natural* in A meaning they can operate in the same way no matter which object they are instantiated with.

An example of monads in **Set** are lists. Given an element $x : A$, then $[x] : \texttt{List}\ A$ is the singleton list. Specifically, $[] : A \to \texttt{List}\ A$ is the operator injecting one element into a list. Moreover, given a list of lists $lls : \texttt{List}\ (\texttt{List}\ A)$ we can obtain a list by concatenation

$$\begin{aligned}
&\texttt{concat} :: \texttt{List}\ (\texttt{List}\ A) \to \texttt{List}\ A \\
&\texttt{concat}\ [] = [] \\
&\texttt{concat}\ (ls : lls) = ls +\!\!+ (\texttt{concat}\ lls)
\end{aligned}$$

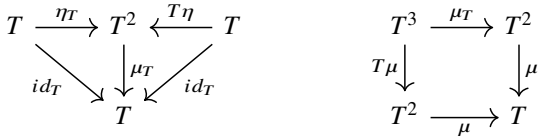
where ++ merges two lists.

From the multiplication one can recover the more familiar bind operator

$$\begin{aligned}
&(>\!>\!=) : \texttt{List}\ A \to (A \to \texttt{List}\ B) \to \texttt{List}\ B \\
&c >\!>\!= f = \mu_A \circ \texttt{fmap}\ f
\end{aligned}$$

We now give the formal definition of a monad.

Definition 5.1 (Monad)**.** *For a category C, a* monad *is an endofunctor $T : C \to C$ such that there exists two natural transformations $\eta : Id \dot{\to} T$ and $\mu : TT \dot{\to} T$ such that the following diagrams hold:*



Example 5.1 (List Monad)**.** *The list type $TX = 1+A\times TX$ is a monad with $\eta_X : X \to TX$ being the map constructing a singleton list and the multiplication $\mu_X : TTX \to X$ given by concatenation applied to lists of lists.*

Example 5.2 (State Monad)**.** *Assume a set of state S and let T :* **Set** $\to$ **Set** *be the* state monad $TX = S \to X \times S$*, that is the monad of computations that take a state $\sigma \in S$ and returns an output in A along with the modified state. The unit of the monad is given be $a \mapsto \lambda\sigma.\langle a, \sigma\rangle$ and the multiplication is given by*

$$c \mapsto \lambda\sigma.c'(\sigma') \text{ where } (c', \sigma') = c(\sigma)$$

For the reader more familiar with functional programming, the multiplication gives raise to the bind operation $>>=_A : TA \to (A \to TB) \to TB$ defined by $\mu_A \circ T(f)$.

5.1 The Computational λ-calculus

The computational λ-calculus which we denote by λ_C is a calculus with effects first devised by Eugenio Moggi [6] who was the first to discover the connection between computational effects and monads.

In this section we define the computational λ-calculus and we give it an interpretation in the Kleisli category C_T for a (strong) monad T.

5.1.1 Syntax

We first define the syntax of λ_C by defining the set of types, the terms and the typing system as usual. To demonstrate the utility of monads we are only going to need basic types and function spaces:

$$\begin{array}{rcll} A, B \in \text{Types} & ::= & \mathsf{unit} & \text{(unit type)} \\ & | & A \to B & \text{(functions)} \end{array}$$

while the set of λ-terms Terms is inductively defined as follows

$$\begin{array}{rcll} t \in \text{Terms} & ::= & x & \text{(terms variables)} \\ & | & \text{bang} & \text{(effects)} \\ & | & \lambda x.t \mid t_1 t_2 & \text{(functions)} \end{array}$$

where bang is a program producing an effect. The typing judgement relation $\vdash$ inductively as follows

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{}{\Gamma \vdash \text{bang} : \mathsf{unit}}$$

$$\frac{\Gamma \vdash t_1 : A \to B \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$$

Note that the program bang returns nothing so we type it with the type unit.

5.1.2 Semantics

For this particular effect we define a store monad T as

$$TA = (\mathbb{N} \times A)^{\mathbb{N}}$$

where $\mathbb{N}$ represents how many bangs have been produced at a given time. Specifically, a computation TA is a function which takes the amount of bangs that have been produced, possibly generates more and returns the final result. The unit of the monad is $\eta_A(x)(n) = \langle n, x\rangle$ and the multiplication is defined as

$$\begin{aligned}\mu_A(c)(z) = \ & \texttt{let } \langle n, c'\rangle \leftarrow c(z) \texttt{ in} \\ & \texttt{let } \langle m, x\rangle \leftarrow c'(n) \texttt{ in} \\ & \langle n + m, x\rangle\end{aligned}$$

Now rather than interpreting programs as arrows in the homset $\mathbf{Set}(A, B)$ we want to interpret programs as arrows $\mathbf{Set}(A, TB)$ to indicate the fact that a program once given an input of type A will produce an effect along with the result of the computation. First we need an identity program. For this we will use the unit of the monad $\eta_A : A \to TA$ which has the right type. Secondly, we need a way of composing to programs. Hence given two programs $f \in \mathbf{Set}(A, TB)$ and $g \in \mathbf{Set}(B, TC)$ we can define the operation $\circ_T$ as

$$f \circ_T g(x) = f(x) >>= g$$

using the bind operation of the monad. This composition is associative and respects the identity programs.

What we obtained is a category where objects are the same objects as in the original category and arrows are maps $A \to TB$. This is called the Kleisli category for a monad T.

Definition 5.2 (Klesli Category)**.** *For a category C and a monad (T, η, μ) the Kleisli category, denoted by C_T, is the category where objects are the objects in C and arrows $f : A \to B$ are arrows $f_T : A \to TB$ in C.*

We would like to stress that when we speak about arrows $f : A \to B$ in the Kleisli category C_T we mean arrows $f_T : A \to TB$ in C.

We now proceed at interpreting our types. Contexts Γ are interpreted as usual using the product :

$$[\![x_1 : A_1, \cdots, x_n : A_n]\!] = [\![A_1]\!] \times \cdots \times [\![A_n]\!]$$

The interpretation of types is then a function $[\![\,]\!] : \text{Types} \to \text{Obj}(C_T)$. Notice that $\text{Obj}(C_T)$ is exactly $\text{Obj}(C)$.

$$\begin{aligned}[\![\mathsf{unit}]\!] &= 1 \\ [\![A \to B]\!] &= T[\![B]\!]^{[\![A]\!]}\end{aligned}$$

The placement of which types can produce an effect is crucial. When dealing with effects it is customary to prefer a call-by-value semantics to avoid that effectful computation

passed onto functions as inputs could be executed more than once to the nature of call-by-name.

Because in call-by-value effects are computed before the β-reduction rule applies there is no need for input values to be computations, they are actually normalised values. However, once an input value is applied to a function, this can produce an effect.

With this in mind we interpret the terms of the language as arrows $\mathbf{Set}_T(\Gamma, A)$ by induction on the typing judgment. For a well-typed term $\Gamma \vdash t : A$ we give an arrow $[\![t]\!]$ of type $[\![\Gamma]\!] \to [\![A]\!]$ in C_T as usual, but here the reader should keep in mind that this is really a map of type $[\![\Gamma]\!] \to T[\![A]\!]$ in C.

$$\begin{aligned}
[\![\Gamma \vdash () : \mathsf{unit}]\!] &= \eta_1 \circ ! \\
[\![\Gamma \vdash \mathrm{bang} : \mathsf{unit}]\!] &= b \circ ! \\
[\![\Gamma \vdash x : A]\!] &= \eta_{[\![A]\!]} \circ \pi_x \\
[\![\Gamma \vdash \lambda x.t : A \to B]\!] &= \eta_{T[\![A]\!]^{[\![B]\!]}} \circ \Lambda[\![t]\!] \\
[\![\Gamma \vdash t_1 t_2 : B]\!] &= \mathrm{app}([\![t_1]\!], [\![t_2]\!])
\end{aligned}$$

A program of type unit is in fact a map $\Gamma \to T1$. This map is obtained by disregarding the context Γ, using $[\![\Gamma]\!] \xrightarrow{!} 1$ and then using the unit of the monad $\eta_1 : 1 \to T1$. The bang operation is interpreted using the semantic bang $bang : 1 \to T1$ defined as $bang(*)(n) = n + 1$. Variables are interpreted using the unit of the monad.

In the case of lambda abstractions, our aim is to produce a map $[\![\Gamma]\!] \to T[\![A \to B]\!]$ which is equal to having a map $[\![\Gamma]\!] \to T(T[\![B]\!])^{[\![A]\!]}$. Using the currying of the exponential we turn a map $[\![t]\!] : [\![\Gamma, A]\!] \to T[\![B]\!]$ to a map $\Lambda[\![t]\!] : [\![\Gamma]\!] \to (T[\![B]\!])^{[\![A]\!]}$. The post-composition with the unit of the monad yields the desired type.

Finally, function application means we have to produce a function $[\![\Gamma]\!] \to T[\![B]\!]$ out of the maps $[\![\Gamma]\!] \to T[\![A \to B]\!]$ and $[\![\Gamma]\!] \to T[\![A]\!]$. Crucially we need the *strength* of a monad. This is a map $st_{A,B} : A \times TB \to T(A \times B)$. In **Set**, every endofunctor F is strong (and thus every monad)

$$st_{A,B}(a, fb) = F(\lambda x.(a, x))(fb)$$

Once we have strength we can define app as the map which makes the below diagram commute.

$$\begin{array}{ccccc}
\Gamma & & \xrightarrow{\qquad\qquad\qquad\qquad\qquad\qquad} & & TB \\
\downarrow{\scriptstyle \langle t_1, t_2 \rangle} & & \xrightarrow[\mathrm{app}(t_1,t_2)]{} & & \uparrow{\scriptstyle \mu_B} \\
T(TB^A) \times TA & & & & T^2B \\
\downarrow{\scriptstyle st_{TB^A,TA}} & & & & \uparrow{\scriptstyle \mu_{TB}} \\
T(TB^A \times TA) & \xrightarrow{T(st_{TB^A,A})} & T(T(TB^A \times A)) & \xrightarrow{T^2(\epsilon)} & T^3B
\end{array}$$

6 Conclusions

In this tutorial we introduced the basic categorical notions of categories, universal constructions, functors, and natural transformations, and saw how they provide a clean

language for describing structure and compositionality.

We have explored the semantics of programming languages by viewing types as sets (or objects) and programs as functions (morphisms) and connected these ideas directly to programming languages. This correspondence explains why the simply typed λ-calculus is naturally interpreted in Cartesian closed categories: the categorical structure matches the type structure of the calculus.

We then extended this semantic viewpoint to Moggi's computational λ-calculus, where monads model computational effects and distinguish values from computations in a principled way.

Index

References

[1] Steve Awodey. *Category Theory*. Oxford University Press, Inc., USA, 2nd edition, 2010.

[2] C.A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of computing. MIT Press, 1992.

[3] Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, New York, 1971. Graduate Texts in Mathematics, Vol. 5.

[4] B. Milewski. *Category Theory for Programmers*. Blurb, Incorporated, 2018.

[5] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[6] Dana S. Scott. A type-theoretical alternative to iswim, cuch, OWHY. *Theor. Comput. Sci.*, 121(1&2):411–440, 1993.

[7] Thomas Streicher. *Domain-theoretic Foundations of Functional Programming*. World Scientific, 2006.

[8] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.

A Elements of Set Theory

A *set* (or class) is an unordered collection of objects called *elements*. If an object a is an element of a set X, we write $a \in X$ and say "a belongs to X."

One of the most fundamental sets is the *empty set*, denoted $\emptyset$, which contains no elements. It is important to distinguish between $\emptyset$ and the *singleton set* $\{\emptyset\}$, which contains the empty set as its only element.

Other notable sets include the set of *natural numbers*, $\mathbb{N} = \{1, 2, 3, \dots\}$, and the set of *natural numbers with zero*, $\mathbb{N}_0 = \{0, 1, 2, 3, \dots\}$.

Given two sets A and B, we can construct their *cartesian product*, $A \times B$, which is the set of all ordered pairs (a, b) such that $a \in A$ and $b \in B$. Similarly, the *union* of two sets A and B, written $A \cup B$, is the set containing all elements of A and B, with duplicates removed. A related concept is the *disjoint union*, $A \uplus B$, which pairs elements with tags to distinguish their origin: $(1, a)$ for $a \in A$ and $(2, b)$ for $b \in B$. Given an equivalence relation $\sim$ on X, the *quotient* of a set X by an equivalence relation $\sim$ is the set of equivalence classes:

$$X/{\sim} = \{[x] \mid x \in X\},$$

where $[x] = \{y \in X \mid x \sim y\}$. The quotient map $q : X \to X/{\sim}$ sends each element $x \in X$ to its equivalence class $[x]$. The quotient set partitions X into disjoint equivalence classes. The union of two sets A and B can be expressed as a quotiented disjoint union:

$$A \cup B = (A \uplus B)/{\sim},$$

where $(1, a) \sim (2, b)$ if and only if $a = b$.

A.1 Relations and Functions

A *relation* $R \subseteq A \times B$ is a subset of the Cartesian product $A \times B$, linking certain elements of A to elements of B. A *function* is a special type of relation $f \subseteq A \times B$ such that for every $a \in A$, there exists exactly one $b \in B$ related to it. The set of all functions from A to B is denoted $A \to B$.

Two special cases of functions are worth noting. First, there is only one function from the empty set $\emptyset$ to any set A, which is the empty relation $! \subseteq \emptyset \times A$. Second, there is only one function from any set A to a singleton set $\{*\}$, which maps every element $a \in A$ to $*$. This function is also denoted $!$, and its meaning is typically clear from context.

An *equivalence relation* on a set X is a relation R that satisfies three properties: *reflexivity*, meaning that every element is related to itself ($\forall x \in X, xRx$); *symmetry*, meaning that if one element is related to another, then the second is related to the first ($\forall x, y \in X, xRy \Rightarrow yRx$); and *transitivity*, meaning that if one element is related to a second and the second to a third, then the first must be related to the third ($\forall x, y, z \in X, xRy$ and $yRz \Rightarrow xRz$).

A *preorder* is a relation that satisfies only reflexivity and transitivity, making it a weaker form of ordering that allows for indistinguishable elements. A *partial order* strengthens this by also requiring *antisymmetry*, which states that if two elements are mutually related, they must be equal ($\forall x, y \in X, xRy$ and $yRx \Rightarrow x = y$).

A.2 Cardinality and Isomorphisms

For two sets X, Y, a *function* $f : X \to Y$ is called *injective* (or one-to-one) if different inputs always produce different outputs, formally expressed as $f(x_1) = f(x_2) \Rightarrow x_1 = x_2$. It is *surjective* (or onto) if every element of Y is mapped to by at least one element of X, meaning $\forall y \in Y, \exists x \in X$ such that $f(x) = y$. When a function is both injective and surjective, it is called a *bijection* (or a one-to-one correspondence), ensuring that every element of X is uniquely paired with an element of Y and vice versa, allowing for the existence of an inverse function $f^{-1} : Y \to X$.

The *size* or *cardinality* of a set measures how many elements it contains. Two sets are said to have the same cardinality, or to be *isomorphic*, if there exists a bijective function $f : A \to B$ with an inverse $f^{-1} : B \to A$ such that $f(f^{-1}(x)) = x$ and $f^{-1}(f(x)) = x$ for all x.

A set A is *finite* if it is isomorphic to the set $\{m \in \mathbb{N} \mid m \leq n\}$ for some $n \in \mathbb{N}$. In this case, we can enumerate its elements as $A = \{a_1, a_2, \ldots, a_n\}$. A set is *infinite* if it is not finite. A set is *countable* if it is isomorphic to the natural numbers $\mathbb{N}$.

A.3 Indexed Families of Sets

For a set I, an *indexed family of sets* is a collection of sets $\{A_i\}_{i \in I}$, where each set A_i is associated with an index $i \in I$. If I is finite, we can write the union and Cartesian product of these sets as:

$$A_1 \cup A_2 \cup \cdots \cup A_n \quad \text{and} \quad A_1 \times A_2 \times \cdots \times A_n.$$

If I is infinite, the *union* of the family is:

$$\bigcup_{i \in I} A_i = \{a \in A_i \mid i \in I\},$$

and the *dependent product* (or *infinite product*) is the set of functions $f : I \to \bigcup_{i \in I} A_i$ such that $f(i) \in A_i$ for all $i \in I$:

$$\Pi_{i \in I} A_i = \{f : I \to \bigcup_{i \in I} A_i \mid f(i) \in A_i\}.$$

A.4 The Russel's Paradox

Russell's Paradox is a fundamental problem in set theory, discovered by Bertrand Russell in 1901. It reveals a contradiction in naive set theory, which allowed the formation of any set based on a defining property, without restrictions. The paradox shows that such a theory can lead to logical inconsistencies.

The paradox arises when we consider the set of all sets that do not contain themselves as a member. Let's define this set as R. Formally, R is the set of all sets that do not contain themselves as a member. In other words:

$$R = \{x \mid x \notin x\}$$

Here, R is the set of all sets x such that x does *not* contain itself as a member.

Now, the central question of the paradox is: **Does the set R contain itself?**

To answer this, we explore two possibilities. The case when $R \in R$ and the case when $R \notin R$. If R is a member of itself, then by the definition of R, it must not contain itself (because R is the set of all sets that do not contain themselves). Therefore, if $R \in R$, it must follow that $R \notin R$, which is a contradiction. If R is *not* a member of itself, then by the definition of R, it must contain itself (because R is the set of all sets that do not contain themselves, and R would be one of those sets). Therefore, if $R \notin R$, it must follow that $R \in R$, which is also a contradiction.

This contradiction shows that the assumption that such a set R can exist leads to an inconsistency. The paradox demonstrates that naive set theory, which allowed for the creation of sets like R, is inherently flawed.

A.4.1 The Axiom of Regularity (Foundation)

The Axiom of Regularity, also known as the Axiom of Foundation, is one of the axioms in Zermelo-Fraenkel set theory (ZF), designed to prevent certain paradoxes like Russell's Paradox.

The Axiom of Regularity states:

$$\forall A \, (A \neq \emptyset \Rightarrow \exists x \in A \, (x \cap A = \emptyset))$$

This means that for every non-empty set A, there exists an element $x \in A$ such that x and A are disjoint sets.

The Axiom of Regularity essentially says that *no set can be a member of itself* (directly or indirectly), and it prevents sets from containing themselves or forming cycles. In other words, it ensures that sets are well-founded, meaning they cannot "loop back" on themselves in any way.

In the case of Russell's Paradox, we defined a set R as:

$$R = \{x \mid x \notin x\}$$

The paradox arose because the set R seemed to both contain itself and not contain itself, depending on the assumption. The Axiom of Regularity helps avoid such contradictions by ensuring that no set can be a member of itself. It guarantees that sets like R, which would allow self-referencing and circular definitions, cannot exist.

Let A be a set, and apply the axiom of regularity to the singleton set containing A, that is $\{A\}$. By the axiom of regularity there must be an element of A which is disjoint from $\{A\}$. Since A is the only element of $\{A\}$, it must be that A is disjoint from $\{A\}$. Therefore, since $A \cap \{A\} = \emptyset$ that means that A does not contain itself by definition of intersection.