A Taste of Categorical Semantics

Marco Paviotti Dominic Orchard

December 19, 2024

Contents

1 Introduction to Category Theory

These notes were written during the postgraduate course on Category Theory at the University of Kent in Canterbury (UK) and they are intended to offer a quick introduction to category theory. Here instead of diving into the most intricate categorical constructions we focus on applications of category theory to semantics of programming languages. We will model the simply typed λ-calculus, predicative polymorphism and λ-calculi with computational effects.

There is a plethora of very well-written books about category which we refer the reader to for a more in-depth introduction on the subject [1, 4].

To learn an abstraction it is almost always advisable to have some concrete notion of the subject we want to study or the abstraction itself will not mean much. For a computer scientist, there are probably two ways to learn category theory, that is through the lenses of mathematics or those of functional programming. While the former is probably the best way the latter is the one that many computer scientists would be more familiar with and in fact certain authors have already preferred to take this path [6].

In these notes we take yet a different approach which is an in-between betwen mathematics and programming languages. In fact, it is the mathematical approach to programming language which goes under the name of *denotational semantics*. The utopic idea is that a type should be regarded as a *set* and a program should be regarded as a *function* between sets. Now of course the more a programming language is equipped with more feature the harder is to maintain this simplicist view. When using category theory to model languages then a type should be an *object* and a program should be an *arrow*. This analogy will be made more precise as we introduce the basic concepts.

We start out with some rudiments of set theory.

1.1 Elements of Set Theory

A *set* (or class) is an unordered collection of objects called *elements* of the set. We write $a \in X$ when a is an element of the set X and we read it as "a belongs to X". Here are some important sets:

- $\emptyset$ the empty set with no elements. The reader should ponder about the difference between the empty set $\emptyset$ and the singleton set $\{\emptyset\}$ containing the empty set.
- $\mathbb{N}$ the set of *natural numbers* $\{1, 2, 3, \dots\}$
- $\mathbb{N}_0$ the set of *natural numbers with zero* $\{0, 1, 2, 3, \dots\}$

- $A \times B$ the *cartesian product* of sets A, B is the set containing the pairs (a, b) such that $a \in A$ and $b \in B$.
- $A \cup B$ the *union* of sets A, B is the set containing all elements of A and of B. If there are equal elements in A, B these become squashed together.
- $A \uplus B$ the *disjoint union* of sets A, B is the set containing all elements $(1, a)$ for $a \in A$ and $(2, b)$ with $b \in B$

A *relation* $R \subseteq A \times B$ is a subset of the cartesian product $A \times B$ relating some elements in A with some elements in B. A *function* is a relation $f \subseteq A \times B$ such that for an $a \in A$ there exists only one $b \in B$. We write the set of functions between A and B as $A \to B$.

There is only one function from the empty set $\emptyset$ into any other set A, that is the empty relation denoted by $! \subseteq \emptyset \times A$. Dually, there is only one function from any set A to the singleton set $\{*\}$ for an element $*$. That is the function sending every $a \in A$ into $*$. We denote this map $!$ as well although it should be clear from the context which one we mean.

An important notion in set theory is the one of *size* which indicates the *cardinality* of a set. Two sets are said to be *isomorphic* when they have the same cardinality. An isomorphism is given by a function $f : A \to B$ and its *inverse* $f^{-1} : B \to A$ such that $f(f^{-1}(x)) = x$ and $f^{-1}(f(x)) = x$.

A set A is *finite* if and only if it is isomorphic to the set $\{m \in \mathbb{N} \mid m \leq n\}$ for some $n \in \mathbb{N}$. If this is the case it means we can enumerate the elements of A and write A as $\{a_1, a_2, a_3, \ldots, a_n\}$. This is because intuitively we could write down its elements on a piece of paper in a finite amount of time. We say a set is *infinite* iff it is not finite. A set is *countable* if it is isomorphic to the natural numbers.

For a set I we denoted the family of sets indexed by I as $\{A_i\}_{i \in I}$. In the case I is finite the family $\{A_i\}_{i \in I}$ is finite and we can write both the finite union and product of these sets as follows:

$$A_1 \cup A_2 \cup \cdots \cup A_n$$

and similarly for the product. In the case I is infinite the union of the infinite family of sets can be written simply as

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

Dually the *infinite product* or *dependent product* is defined by the set of functions that given an index $i \in I$ return an element in A_i.

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

1.2 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[1]

[1]Disclaimer: category theory was of course invented way before we started giving semantics of programming languages by Samuel Eilenberg and Saunders MacLane.

Definition 1.1 (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* $id_A : A \to A$ *such that and for arrows* $f : A \to B$ *and* $g : B \to C$ *we there exists an arrow* $g \circ f : A \to C$ *such that the identity and associativity laws hold:*

$$id_A \circ f = f \circ id_A = f$$
$$f \circ (g \circ h) = (f \circ g) \circ h$$

Example 1.1. *The category where objects are sets and arrows are functions between them is denoted by* **Set**.

1.3 Initial and terminal objects

The initial object is an object denoted by 0 such that for every other object A there exist a unique arrow $0 \to A$. This is called the universality property of the object 0. We draw a drashed arrow as follows to indicate the arrow is unique:

$$0 \dashrightarrow A$$

Similarly, the *terminal* object is denoted by 1 and is such that for every other object A there is a unique arrow $A \to 1$

$$A \dashrightarrow 1$$

It is important to know that such objects in category theory are unique only up-to isomorphism.

Let us look at an example to see this. In **Set**, the initial object is the empty set $\emptyset$ since there is a unique function from the empty set to any other set A, that is the empty function. Similarly, the terminal object is any singleton set since for any singleton set, say the set $\{*\}$, for a given set A there is a unique function in to $\{*\}$ which is the constant function mapping every element $x \in A$ into $*$. Notice that there are many terminal object in category, but they are all isomorphic in that they all contain only one element.

1.4 Isomorphisms

Category theory abstracts the notion of isomorphism.

Definition 1.2 (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 is the correct definition of isomorphism in the following sense.

As per the categorical definition, 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).

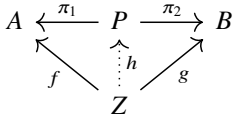
1.5 Cartesian Closed Categories

In this section we will introduce the basic building blocks to be able to interpret the simply typed λ-calculus (STLC). We first introduce products and exponentials.

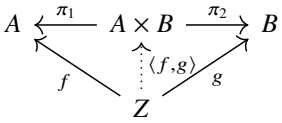
1.5.1 Products

We proceed generalise the set-theoretical concept of cartesian product $A \times B$.

Definition 1.3 (Product)**.** *For two objects A and B the* product *is an object P quipped 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:*



There is a notational convention where we write $A \times B$ for the product of two objects A and B and $\langle f, g \rangle$ for h which means we can rewrite the diagram as follows:



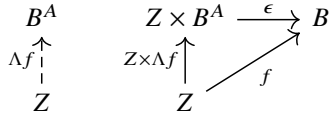
Proposition 1.1 (Products are unique up to isomorphism)**.** *That is, if we have a category with two notions of product, then there is an isomorphism between these products.*

Exercise 1.1. *Prove Proposition 1.1. Hint: you have to prove that for two objects A and B if you have two products P and P' for the same two objects you can derive an isomorphism. This is constructed by using the universality property of products.*

1.5.2 Exponentials

In this section we generalise the idea of function space between two sets.

Definition 1.4 (Exponentials)**.** *An* exponential *is an object denoted by B^A which has an arrow $\epsilon : A \times B^A \to B$ called the* evaluation map *which is such that for every arrow $f : Z \to B$ there exists a unique arrow, denoted by $\Lambda f : Z \to B^A$ and pronounced the* currying *of a function, such that the following diagram commutes*



1.6 Semantics of the Simply Typed λ-Calculus

In this section we look at a categorical semantics for the Simply-Typed λ-calculus (STLC)[5].

1.6.1 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

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

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

$$\begin{array}{rcll} t \in \text{Terms} & ::= & x & \text{(terms variables)} \\ & | & \underline{n} & \text{(natural numbers)} \\ & | & \lambda x.t \mid t_1 t_2 & \text{(functions)} \\ & | & (t_1, t_2) \mid \pi_1(t) \mid \pi_2(t) & \text{(products)} \end{array}$$

and finally, a typing judgement relation $\vdash$ inductively as follows

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

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_1(t) : A} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_2(t) : B} \qquad \frac{\Gamma \vdash t_1 : A \quad \Gamma \vdash t_2 : B}{\Gamma \vdash \langle t_1, t_2 \rangle : A \times B}$$

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

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

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

And, finally, the function $[\![\cdot]\!] : \text{Terms} \to \text{Arr}(\mathcal{C})$ interprets a term as a morphism between two objects in the category $\mathcal{C}$. 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:

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

which is notation

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

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

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.

1.7 Opposite categories

Definition 1.5 (Opposite category)**.** *For a category C, there is a category C^{op} which has as objects the same objects as C and for every morphism $f : A \to B$ in C a morphism $f^{\mathsf{op}} : B \to A$*

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

Example 1.2 (Is $\mathbf{Set} \cong \mathbf{Set}^{\mathsf{op}}$?)**.** *Whenever we have an isomorphism $f : X \to Y$, every property that holds for X should hold for Y and viceversa.*

Now assume we have an isomorphism (of categories) $F : \mathbf{Set} \to \mathbf{Set}^{\mathsf{op}}$ and consider a map $f : A \to \emptyset$ in **Set***. (You should read about initial and terminal objects to understand what $\emptyset$ is).*

Since $\emptyset$ is initial in **Set** *then $F\emptyset$ should be terminal in $\mathbf{Set}^{\mathsf{op}}$. Now $Ff : FA \to F\emptyset$ is a map in $\mathbf{Set}^{\mathsf{op}}$ and because $F\emptyset$ is terminal we have many more arrows $FA \to F\emptyset$ than $A \to \emptyset$. Exercise: Work this out!.*

Example 1.3 (Is $\mathbf{Rel} \cong \mathbf{Rel}^{\mathsf{op}}$?)**.** *First off,* **Rel** *has as objects sets X, Y, Z and an arrow $f : X \to Y$ is a relation $f \subseteq X \times Y$. Define now two functors $F :$ **Rel** $\to \mathbf{Rel}^{\mathsf{op}}$ and $G : \mathbf{Rel}^{\mathsf{op}} \to$ **Rel** such that they are one the inverse of the other. We define $FX = X$ and $F(R : X \to Y)(y \in Y) = \{x \mid x\ R\ y\}$ and $GX = X$ and $G(R : Y \to X)(y \in Y) = \{x \mid y\ R\ x\}$*

Verify that this is an isomorphism. (You first need to define the composition of two morphisms (relations) in **Rel***)*

2 Functors and Natural Transformations

Definition 2.1 (Functor)**.** *Let C and $\mathcal{D}$ be two categories. A* functor *$F : C \to \mathcal{D}$ is a mapping between categories that associates objects in C to an object $FC \in Obj(C)$ 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:*

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

Every functor preserves isomorphisms:

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

A functor whose functorial action is surjective is called *full*. whilst when the functorial action is injective the functor is called *faithful*.

Proposition 2.1. *A fully faithful functor F preserves and reflects isomorphisms:*

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

Exercise 2.1. *Prove Proposition 2.1*

Definition 2.2 (Natural Transformation)**.** *For two functors $F, G : \mathcal{C} \to \mathcal{D}$, 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:*

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

2.1 The Homset Functor

Given a (locally small) category $\mathcal{C}$[1] the homset $\hom_{\mathcal{C}}(A, B)$, for any two objects A, B, induces two functors.

The first is the covariant functor $\hom_{\mathcal{C}}(A, -) : \mathcal{C} \to \mathbf{Set}$ which sends an object B to the set of arrows between A and B and an arrow $B \to B'$ to the functorial action $\hom_{\mathcal{C}}(A, f) : \hom_{\mathcal{C}}(A, B) \to \hom_{\mathcal{C}}(A, B')$ which sends an arrow $g : A \to B$ to the post-composition $f \circ g : A \to B'$.

The second is the contravariant functor $\hom_{\mathcal{C}}(-, B) : \mathcal{C}^{\mathsf{op}} \to \mathbf{Set}$ which sends an object A to the set of arrows between A and B. This functor is contravariant because it takes an arrow $A \to A'$ to the functorial action $\hom_{\mathcal{C}}(f, B) : \hom_{\mathcal{C}}(A', B) \to \hom_{\mathcal{C}}(A, B)$ which sends an arrow $g : A' \to B$ to the pre-composition $g \circ f : A \to B$.

2.1.1 Natural Isomorphisms

A natural isomorphism is an *isomorphism of functors*. We the tools provided by category theory this should not be a hard notion to derive since we can form the category of functors $\mathcal{C}^{\mathcal{D}}$ and natural transformations betwen them. At this point an isomorphism in this category is precisely an isomorphism of functors.

More precisely, for two functors $F, G : \mathcal{C} \to \mathcal{D}$ we say these two functors are isomorphism if there exist a natural transformation $\phi : F \dot{\to} G$ which has an inverse $\phi^{-1} : G \dot{\to} F$ which is also a natural transformation.

Example 2.1. *For example, the stream functor $Str\ \cdot : \mathbf{Set} \to \mathbf{Set}$ defined by the greatest solution to the following equation*

$$Str\ A \cong A \times Str\ A \tag{1}$$

is isomorphic to the functor $\hom_{\mathbf{Set}}(\mathbb{N}, -)$, i.e. the following isomorphism is natural in A

$$Str\ A \cong \hom_{\mathbf{Set}}(\mathbb{N}, A) \tag{2}$$

Exercise 2.2. *Prove this fact by defining a natural transformation $\phi_A : Str\ A \to \hom_{\mathbf{Set}}(\mathbb{N}, A)$ and its inverse.*

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

In general, any covariant functor F that is isomorphic to the hom functor $\hom_C(A, -)$ for some $A \in \mathrm{Obj}(()C)$ is called *representable*.

Exercise 2.3. *Show that the type of lists as defined below*

$$ListA \cong 1 + A \times ListA \tag{3}$$

is not representable.

3 Limits and Colimits

We are going to proceed up the ladder of abstraction and generalise the notion of arbitrary products and coproducts.

Given a set I and a family of sets indexed by I, $\{A_i\}$ we can form the *dependent product* of these sets $\Pi_{i \in I} A_i$ which informally is the product of all sets A_i

$$A_1 \times A_2 \times A_3 \times \cdots \times A_n \times \ldots$$

Dually we can form the *dependent sum* is denoted by $\sum_{i \in I} A_i$ and can be written informally as follows:

$$A_1 + A_2 + A_3 + \cdots + A_n + \ldots$$

Limits and Colimits are just generalisation of these concepts.

Definition 3.1 (Cone)**.** *For a diagram $D : \mathcal{I} \to \mathcal{D}$, a* cone *is an object C such that ther exists a family of maps $\pi_i : C \to D_i$ and such that*

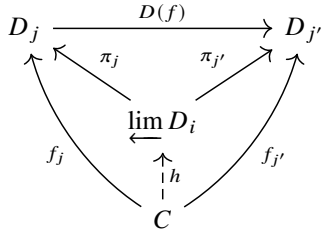
$$D(f) \circ \pi_i = \pi_j \tag{4}$$

for every $f : i \to j$.

Definition 3.2 (Limit)**.** *For a diagram $D : \mathcal{I} \to C$, a* limit *is a universal cone P, that is a cone that for any other cone for D, say C, there exists a unique arrow $h : C \to \lim_{i \in \mathcal{I}} D_i$ such that*

$$\pi_i \circ h = f_i \tag{5}$$

for every $i \in \mathcal{I}$. The picture below summarises the situation:



We denote the limit of a diagram D by $\varprojlim_{i \in \mathcal{I}} D_i$

Exercise 3.1. *The notion of colimit is obtained by reversing the arrows. Derive it by exercise.*

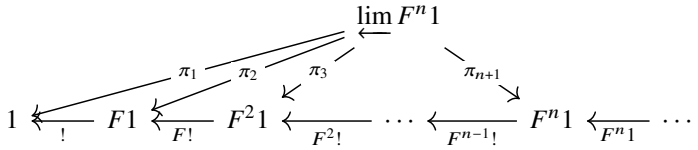
Dependent Products and Sums The dependent product and dependent sum are just limits and colimits respectively where the category $\mathcal{I}$ is discrete[1]. In this particular case the coherence conditions (4) on the projections are vacuous because there are no meaningful arrows in the index category. This means, for example in the case of the product, that every tuple is in the limit as opposed to just those that satisfy the coherence condition.

Powers and Copowers Assume that our diagram $D : \mathcal{I} \to C$ is constant additionally to the category $\mathcal{I}$ being discrete and assume $D(i) = A$ for some $A \in C$. Then we have another special case where the dependent product becomes a product of the same object $\Pi_{i \in \mathcal{I}} A$ which is called the *power* and denoted by $A^{\mathcal{I}}$. Note that this is in general different from the exponential because $\mathcal{I}$ is a category, not an object. However, in **Set** the power is isomorphic to the function space $\mathcal{I} \to A$ since $\mathcal{I}$ can be viewed as a set because it is discrete[1].

Similarly, the *copower* is a special case of dependent sum when D is constant and is denoted by $\mathcal{I} \bullet A$. Moreover in **Set**, this is just the pair $\mathcal{I} \times A$ with $\mathcal{I}$ again viewed as a set.

3.1 Algebraic Data Types as Limits and Colimits

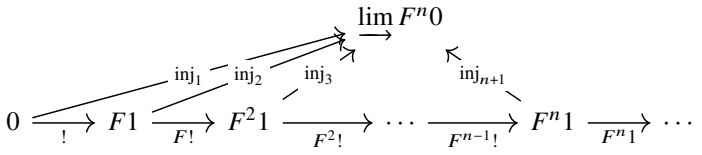
Streams as Limits Assume C is a category with all limits. The streams over A from definition (1) are obtained as the limit for a diagram. We first define the functor $F : \mathbf{Set} \to \mathbf{Set}$ by $FX = A \times X$ giving the non-recursive shape of the streams. Then we define the ω^{op}-chain of approximations represented by the diagram $D : \omega^{\mathsf{op}} \to C$ defined on object by $D(1) = 1$ and $D(n+1) = F^n 1$ and on arrows by $D(1 \leq 2) =!$ (the unique map to the terminal object) and $D(n \leq n+1) = F^n !$. The limit of this ω^{op}-chain is the type of coinductive streams as depicted below:



Lists as Colimits Assume C is a category with all colimits. The lists over A from definition (3) are obtained as the colimit for a diagram. We first define the functor $F : \mathbf{Set} \to \mathbf{Set}$ by $FX = 1 + A \times X$ giving the non-recursive shape of the lists. Then we define the ω-chain of approximations represented by the diagram $D : \omega \to C$ defined on object by $D(1) = 0$ and $D(n+1) = F^n 0$ and on arrows by $D(1 \leq 2) =!$ (the unique map from the initial object) and $D(n \leq n+1) = F^n !$. The colimit of this ω-chain is the type of coinductive streams as depicted below:

[1]The category of objects and identity arrows

[1]Notice that $\mathcal{I}$ needs to be small for the collection of objects to be a set, but again, size issues are not really an issue for the focus of these notes



4 Adjunctions

Almost every universality property comes from an adjunction[1] and certainly all the constructions seen so far are in fact an instance of an adjunction.

4.0.1 Adjunctions

Given two functors $L : \mathcal{D} \to C$ and $R : C \to \mathcal{D}$ and *adjunction* is an isomorphism of homsets

$$\lfloor \cdot \rfloor : C(LA, B) \cong \mathcal{D}(A, RB) : \lceil \cdot \rceil$$

which is furthermore natural in A and B. Here $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ are the maps witnessing the isomorphism. The adjunction is usually depicted as follows

$$C \underset{R}{\overset{L}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathcal{D}$$

We say that that L is *left adjoint* to R, and viceversa, R right adjoint to L. However, you draw the adjunction the turnstile points towards the left-adjoint as indicated by $L \vdash R$.

As a consequence of the isomorphism we have that for all $f : LA \to B$ and $g : A \to RB$ there is a correspondence of arrows:

$$\lfloor f \rfloor = g \iff f = \lceil g \rceil$$

moreover, the natural isomorphism gives rise to the *fusion laws*. For $a : A' \to A$, $b : B \to B'$, $f : LA \to B$ and $g : A \to RB$

$$R(b) \cdot \lfloor f \rfloor = \lfloor b \cdot f \rfloor \tag{6}$$
$$\lfloor f \rfloor \cdot a = \lfloor f \cdot L(a) \rfloor \tag{7}$$
$$b \cdot \lceil g \rceil = \lceil R(b) \cdot g \rceil \tag{8}$$
$$\lceil g \rceil \cdot L(a) = \lceil g \cdot a \rceil \tag{9}$$

This is really all about adjunctions. All the other definitions and constructions are equivalent to this one. Furthermore, this material is very well covered elsewhere (e.g. in Awodey's book [1]) so we will not be covering it further.

[1] More in general a universal map is the initial object in the comma category $X \downarrow F$, but that is not our concern here.

4.1 Instances of Adjunctions

4.1.1 Initial and Terminal Objects

Assume that we have a category **1** with only one object $*$ and one arrow, the identity arrow id_*. Then the universality property of the initial and terminal object can be then rephrased in terms of adjunctions

$$C \underset{\Delta}{\overset{0}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathbf{1} \qquad \mathbf{1} \underset{1}{\overset{\Delta}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} C$$

where 0 is the constant functor returning the initial object and 1 is the functor returning the terminal object while Δ is the constant functor returning the element $*$.

We can prove that if the above are indeed adjunctions then the functors 0 and 1 must given the initial and terminal object respectively. The isomorphism given by the adjunction with 0 as left adjoint is as follows:

$$\lfloor \cdot \rfloor : \hom_C(0, Y) \cong \hom_{\mathbf{1}}(*, *) : \lceil \cdot \rceil$$

while for the terminal object the adjunction is given by the following natural isomorphism:

$$\lfloor \cdot \rfloor : \hom_{\mathbf{1}}(*, *) \cong \hom_C(X, 1) : \lceil \cdot \rceil$$

Exercise 4.1. *Compute the two naturality conditions and derive the fusion laws.*

We now proceed with the universal property of products and coproducts.

4.1.2 (Co)products and Products

Similary the coproduct arises as the left adjoint of the diagonal functor $\Delta : C \to C \times C$ which is defined as $\Delta X = (X, X)$ while the product arises as the right adjoint of the functor Δ:

$$C \underset{\Delta}{\overset{+}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} C \times C \qquad C \times C \underset{\times}{\overset{\Delta}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} C$$

Exercise 4.2. *Derive the fusion laws and conclude these are exactly those of the product and coproduct respectively.*

4.1.3 Exponentials

The exponential is given by the right adjoint of the functor $(- \times A)$. That is the functor $(-)^A$:

$$C \underset{(-)^A}{\overset{(-)\times A}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} C$$

The isomorphism induced by this adjunction is stated as follows:

$$\lfloor \cdot \rfloor : \hom_C(X \times A, Y) \cong \hom_C(X, Y^A) : \lceil \cdot \rceil$$

Notice that here $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ are respectively the curry and uncurry operations in functional programming.

4.1.4 (Co)Limits

Limits and colimits stem from adjunctions too.

The limit, in particular, extends to a functor from the category of diagrams C^I to the category C simply taking a diagram and returning its limit. This functor is right adjoint to the diagonal functor $\Delta : C \to C^I$ defined as $\Delta X I = X$ mapping an object to the constant diagram which returns that object. Dually, colimits are left-adjoints to the diagonal functor. The situation is depicted below:

$$C \underset{\Delta}{\overset{\varinjlim}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} C^I \underset{\varprojlim}{\overset{\Delta}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} C$$

At this point the reader should notice that the similarity between initial and terminal, coproducts and products and colimits and limits. Indeed initial objects and coproducts are special cases of colimits while terminal objects and products are special cases of limits.

4.2 Semantics of Predicative Polymorphism

Polymorphism is a core feature of most programming languages allowing developers to craft generic programs which are agnostic to the specifics of the types.

In programming languages like C, we can define algorithms that work for lists that contain different types, but we have to define the algorithm for each and everyone of the specific type we want to handle. This type of polymorphism is called *ad-hoc polymorphism*. *Predicative polymorphism* allows the programmer to write one function that is *parametric* on the type or, in words, it takes a generic type with the promise that the data of that type will not be inspected. Therefore, this kind of algorithm can be said to be agnostic as to the type of data structure given. This allows for code reusability since a polymorphic program can be written only once and can work at all types.

This flexibility provides a layer of confidence regarding the properties of the code. Philip Wadler's paper [11] offers a comprehensive exploration of these properties.

As an example, in a total language there is no program of type $\forall\alpha.\alpha$ since such a program would need to yield a value of an arbitrary type α. Consequently, $\forall\alpha.\alpha$ can be viewed as the empty type 0, which, in a total language[1], acts as the initial object.

Similarly, $\forall\alpha.\alpha \to \alpha$ has only one inhabitant, namely the identity function $\Lambda\alpha.\lambda x.x$, with Λ abstracting a type variable and λ abstracting a term variable.

Now the `reverse` function, reversing a list, is indeed a polymorphic function since it needs not inspecting the content of the single element in a list:

$$\forall\alpha.\text{List } \alpha \to \text{List } \alpha \tag{10}$$

This type grants universality by reversing elements without inspecting their specifics. Conversely, any sorting algorithm on lists needs to know that the inner type inside the list as some kind of ordering associated with it, rendering (10) unsuitable for quicksort or mergesort, for instance.

[1] There are of course issues with non-termination but we will not address them here

Once we defined a polymorphic function, we have the liberty to instantiate it with any desired type, even itself. This style of polymorphism is called *impredicative* and was first introduced by Girard in 1972 [2] in the context of proof theory and then proposed by Reynolds in 1983 [9] as a programming language known was *System F* or *second-order λ-calculus*.

For instance, the `reverse` function might operate on elements of type $\mathbb{N}$ or η, but also on the type (10) itself.

While impredicative polymorphism is convenient in programming languages, it poses a significant foundational problem when seeking for a semantic model as we did in Section 1.6. As we have seen the task of seeking a denotational model is to find a universe of sets $\mathcal{U}$ and interpreting types with n free variables as maps $\mathcal{U}^n \to \mathcal{U}$. Assuming no free variables, we would like to interpret $\forall\alpha.\alpha$ as a set. Naively we could try to interpret it as the product of sets indexed over sets. Now, the denotation of $\forall\alpha.\alpha$ would be a set for the interpretation to work and, as a result, we would have that the product of all sets would be the *set containing all sets* which is a paradoxical statement known as the Russell's paradox, suggesting that such a set cannot exist. This fact was discovered by Reynolds in 1984 [10].

Models of impredicative polymorphism have eventually been found by Pitts [8] who showed this in constructive set theory.

In these notes, we are going to avoid this problem and restrict ourselves to *predicative polymorphism* which is a variant of impredicative polymorphism where polymorphic types (*polytypes* or *type schemes*) can only be instantiated with non-polymorphic types (*monotypes*).

4.2.1 Syntax

The language we are going to define is called ML_0 . We first define the syntax by defining the set of monotypes, the set of polytypes, the set of terms and the typing judgment relation. The set of types monotypes and polytypes is defined inductively as follows:

$$
\begin{array}{rcll}
A, B \in \mathrm{Types}_0 & ::= & \alpha & \text{(type variables)} \\
& | & \mathsf{unit} & \text{(unit type)} \\
& | & \mathsf{nat} & \text{(natural numbers)} \\
& | & A \times B & \text{(products)} \\
& | & A \to B & \text{(functions)} \\
\tau \in \mathrm{Types}_1 & ::= & A \mid \forall\alpha.\tau & \text{(predicative polymorphism)}
\end{array}
$$

Notice that polytypes are of the form $\forall\alpha.\forall\beta.A$ where A is a monotype. Thus types of the form $\forall\alpha.(\forall\beta.\beta) \to \alpha$ for example are not allowed.

The set of λ-terms Terms is inductively defined as follows:

$$
\begin{array}{rcll}
t \in \mathrm{Terms} & ::= & x & \text{(terms variables)} \\
& | & () & \text{(unit)} \\
& | & \underline{n} & \text{(natural numbers)} \\
& | & (t_1, t_2) \mid \pi_1(t) \mid \pi_2(t) & \text{(products)} \\
& | & \lambda x.t \mid t_1 t_2 & \text{(functions)} \\
& | & \texttt{let}\ x = t_1\ \texttt{in}\ t_2 \mid \Lambda\alpha.t \mid t@A & \text{(polymorphism)}
\end{array}
$$

where Λ is a binder which abstracts over type variables, the `let` binds a program M of a polytype inside a program N which returns a monotype. This allows the programmer to write programs such as

$$\texttt{let}\ x = id\ \texttt{in}\ x@\mathsf{unit} : \mathsf{unit} \to \mathsf{unit}$$

where $id : \forall\alpha.\alpha \to \alpha$ and the constructor $M@A$ is the application for terms that have a polytype as a type.

Note that since we have defined types using to separate layers for grammars we can constrain the types that we are going to apply to polymorphic programs. In fact, in the constructor $M@A$ only monotypes are allowed.

We first define a judgment for the contexts and the types. Technically, there are two judgments for types, one for the monotypes and one for the polytypes, but we adopt the same notation for both of them as it should be clear from the context which judgment we mean. First a type context is a list of variables. Theunit type can be typed in any well-formed context and and a type variable can by typed in any well-formed context that contains it. Finally the polymorphic types $\forall\alpha.\tau$ are well-typed if the body τ is open in α and well-typed: The context for term variables is instead a list of pairs $x : \tau$ implying that variables are of poly-typed and therefore can also be mono-typed.

$$\Gamma = (x_1 : \tau_1, \ldots, x_n : \tau_n)$$

We now define the typing judgment for terms. Again, there are technically two typing judgments, the first for mono-typed programs and the second for poly-typed programs. Once again, it should be clear from the context which judgment we are using:

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash () : \mathsf{unit}} \qquad \frac{}{\Gamma \vdash \underline{n} : \mathsf{nat}}$$

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_1(t) : A} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_2(t) : B} \qquad \frac{\Gamma \vdash t_1 : A \quad \Theta \mid \Gamma \vdash t_2 : B}{\Gamma \vdash \langle t_1, t_2\rangle : A \times B}$$

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

$$\frac{\Gamma \vdash t_1 : \tau \qquad \Gamma, x : \tau \vdash t_2 : A}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : A}$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \Lambda\alpha.t : \Pi\alpha.\tau}\text{if } \alpha \notin \mathrm{Ftv}(\Gamma) \qquad \frac{\Gamma \vdash t : \Pi\alpha.\tau}{\Gamma \vdash t@A : \tau[A/\alpha]}$$

The first typing rule is for variables. Notice once again that variables can be typed with a poly-type and therefore the context should accommodate that. Additionally the polytype itself has to be well-typed in the context Θ.

We now justify some choices behind the type system.

First notice the variable rule which says that variables' types can be open. The first example that justifies this is the polymorphic identity function which is typed as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \alpha \vdash x : \alpha}}{\vdash \lambda x.x : \alpha \to \alpha}}{\vdash \lambda x.x : \Pi\alpha.\alpha \to \alpha} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash id : \Pi\alpha.\alpha \to \alpha}}{\Gamma \vdash id : (\alpha \to \alpha) \to \alpha \to \alpha} \qquad \dfrac{\dfrac{}{\Gamma \vdash id : \Pi\alpha.\alpha \to \alpha}}{\Gamma \vdash id : \alpha \to \alpha}}{\Gamma \vdash id\ id : \alpha \to \alpha}}{\vdash \texttt{let}\ id = \lambda x.x\ \texttt{in}\ id\ id : \alpha \to \alpha}$$

4.2.2 Semantics

First of all we have to define the semantics of types. Since types can be open with type variables a type needs to be a mapping from a type environment to the set of types. A type environment is itself a map from the type variables to their set. However, as there is no set of sets we have to define a set containing only those sets are *small*. We call this set the *universe* of small sets and we denoted by $\mathcal{U}$. This set contains the singleton set and the set of natural numbers and it is closed under exponentials. To do this we define $\mathcal{U}_0 = \{1, \mathbb{N}\}$ and $\mathcal{U}_{n+1} = \{X^Y \mid X, Y \in \mathcal{U}_n\} \cup \mathcal{U}_n$ then the universe of monotypes is defined as $\mathcal{U} = \bigcup_{n\in\omega} \mathcal{U}_n$. We denote by $\mathcal{U}^n$ the n-fold product

$$\mathcal{U}^n \triangleq \underbrace{\mathcal{U} \times \cdots \times \mathcal{U}}_{n \text{ times}}$$

Now the interpretation of a monotype A with free type variables $\mathrm{Ftv}(A)$ is a mapping $[\![A]\!] : \mathcal{U}^{|\mathrm{Ftv}(A)|} \to \mathcal{U}$ where $|\mathrm{Ftv}(A)|$ is the number of free type variables in the monotype A. Thus given a semantic type environment $\iota \in \mathcal{U}^{|\mathrm{Ftv}(A)|}$ taking type variables to semantic monotypes we can define the semantics of types by induction on the as follows:

$$\begin{aligned}
[\![\alpha]\!]_\iota &= \iota(\alpha) \\
[\![\mathsf{unit}]\!]_\iota &= 1 \\
[\![\mathsf{nat}]\!]_\iota &= \mathbb{N} \\
[\![A \times B]\!]_\iota &= [\![A]\!]_\iota \times [\![B]\!]_\iota \\
[\![A \to B]\!]_\iota &= [\![B]\!]_\iota^{[\![A]\!]_\iota}
\end{aligned}$$

The semantics of polytypes are interpreted in a bigger universe such that $\mathcal{U}$ is contained in it. The category of sets would do as opposed to the Von Neumann universe used in Gunter's book [3]. We denote the inclusion functor by $J : \mathcal{U} \hookrightarrow \mathbf{Set}$. Now for a polytype τ as a map $[\![\tau]\!] : \mathcal{U}^{|\mathrm{Ftv}(\tau)|} \to \mathbf{Set}$. Specifically, we interpret the universal quantifier as the limit over the monotypes in $\mathcal{U}$. This is a Π-type since the universe $\mathcal{U}$ is a set with no arrows and therefore can be regarded as a discrete category as we have shown in Section 3.

$$[\![\Pi\alpha.\tau]\!]_\iota = \Pi_{X\in\mathcal{U}}[\![\tau]\!]_{\iota[\alpha\mapsto X]}$$

Recall that the Π-type is right adjoint to the diagonal functor, in this instance the base category is $\mathbf{Set}^{\mathcal{U}^n}$ with $n+1$ being the free type variables in τ.

$$\mathbf{Set}^{\mathcal{U}^n \mathcal{U}} \underset{\Pi}{\overset{\Delta}{\underset{\longrightarrow}{\longleftarrow}}} \mathbf{Set}^{\mathcal{U}^n}$$

Notice that $[\![\tau]\!]$ lives in the category $\mathbf{Set}^{\mathcal{U}^n \mathcal{U}}$ which is the same as the category $\mathbf{Set}^{\mathcal{U}^{n+1}}$ of semantic types on $n+1$ variables.

The isomorphism induced by the adjunction is therefore as follows:

$$\lfloor\cdot\rfloor : \mathbf{Set}^{\mathcal{U}^{n+1}}(\Delta\Gamma, \tau) \cong \mathbf{Set}^{\mathcal{U}^n}(\Gamma, \Pi\tau) : \lceil\cdot\rceil \tag{11}$$

Notice that the homset in the category $\mathbf{Set}^{\mathcal{U}^n}$ is the set of natural transformations between functors of type $\mathcal{U}^n \to \mathbf{Set}$.

We now have to define the semantics of a context for term variables $\Gamma \equiv x_1 : \tau_1, \ldots, x_n : \tau_n$. Assume m is the number of free type variables in Γ. Then $[\![\Gamma]\!]$ is a object in $\mathbf{Set}^{\mathcal{U}^m}$:

$$[\![\Gamma]\!] = [\![T_1]\!] \times \cdots \times [\![T_n]\!]$$

where the product is the defined point-wise as $(X \times Y)(\iota) = X_\iota \times Y_\iota$ for a type environment $\iota \in \mathcal{U}^m$.

The interpretation of well-typed terms $\Gamma \vdash t : \tau$ is an arrow $[\![\Gamma \vdash t : \tau]\!] : [\![\Gamma]\!] \xrightarrow{[\![t]\!]} [\![\tau]\!]$ in $\mathbf{Set}^{\mathcal{U}^n}$ whereas a well-typed term $\Gamma \vdash t : A$ is an arrow $[\![\Gamma \vdash t : A]\!] : [\![\Gamma]\!] \xrightarrow{[\![t]\!]} [\![A]\!]$. The interpretation is then given by induction on the typing judgment:

$$\begin{aligned}
[\![\Gamma \vdash () : \mathsf{unit}]\!] &= ! \\
[\![\Gamma \vdash \underline{n} : \mathsf{nat}]\!] &= n \\
[\![\Gamma \vdash x : A]\!] &= \pi_x \\
[\![\Gamma \vdash \mathrm{prj}_1(t) : A]\!] &= \pi_1 \circ [\![t]\!] \\
[\![\Gamma \vdash \mathrm{prj}_2(t) : B]\!] &= \pi_2 \circ [\![t]\!] \\
[\![\Gamma \vdash (t_1, t_2) : A \times B]\!] &= \langle [\![t_1]\!], [\![t_2]\!] \rangle \\
[\![\Gamma \vdash \lambda x.t : A \to B]\!] &= \Lambda [\![t]\!] \\
[\![\Gamma \vdash t_1 t_2 : B]\!] &= \epsilon \circ \langle [\![M]\!], [\![N]\!] \rangle \\
[\![\Gamma \vdash \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 : A]\!] &= [\![t_2]\!] \circ \langle id_\Gamma, [\![t_1]\!] \rangle \\
[\![\Gamma \vdash \Lambda\alpha.t : \forall\alpha.\tau]\!] &= \lfloor [\![t]\!] \rfloor \\
[\![\Gamma \vdash t@A : \tau[A/\alpha]]\!] &= \pi_A \circ [\![t]\!]
\end{aligned}$$

Most of the terms are interpreted as in Section 1.6. Assuming a map $[\![t]\!] : \Delta[\![\Gamma]\!] \to [\![\tau]\!]$ in $\mathbf{Set}^{\mathcal{U}^{n+1}}$ $\alpha \notin \mathrm{Ftv}(\Gamma)$ the interpretation of the introduction rule of the $\forall$ quantifier is given by the adjunction (11) defined above which is an arrow of type

$$[\![\Gamma]\!]_\iota \to \Pi_{X \in \mathcal{U}} [\![\tau]\!]_{\iota[\alpha \mapsto X]}$$

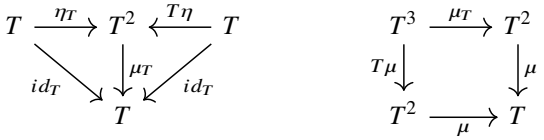
for $\iota \in \mathcal{U}^n$.

5 Monads

In this section we are going to take a closer look at computational effects and how they can be interpreted into a category. To do this need the notion of monad. Here we assume the reader has at least heard of what a monad is from functional programming. Monads in category theory are the same concept, but originally they have been given a different definition.

Definition 5.1 (Monad)**.** *For a category C, a* monad *is an endofunctor $T : C \to C$ such that there exists two natural transformations $\eta : Id \xrightarrow{\cdot} T$ and $\mu : TT \xrightarrow{\cdot} T$ 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 $\mu_A \circ T(f)$.

5.0.1 Adjunctions determine Monads

Given a pair of adjoint functors $L \vdash R$ we can construct both a monad, given by RL and a comonad, given by LR. The unit of the monad and the counit of the comonad are defined as follows

$$\eta_A = \lfloor id_{LA} \rfloor$$
$$\eta_B = \lceil id_{RB} \rceil$$

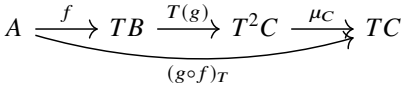
The join or multiplication of the monad $\mu : RLRL \to RL$ is defined as $\mu = R\epsilon_L$ and the cojoin or comultiplication $\delta : LR \to LRLR$ is defined as $\delta = L\eta_R$. The operations of the comonad are dually defined.

5.1 The Kleisli Category

The Kleisli category is the category of arrows that produce an effect 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 have to prove that C_T is a category. This is easy to check as for every object A we have and identity arrow $id_A : A \to A$ given by the unit of the monad $\eta_A : A \to TA$ and for arrows $f : A \to B$ and $g : B \to C$ in C_T we can construct the composite $g \circ f : A \to C$ using the functorial action of the monad $T(g)$ and the multiplication of the monad μ_C:



5.2 The Computational λ-calculus

The computational λ-calculus which we denote by λ_C is a calculus with effects first devised by Eugenio Moggi [7] 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.

5.2.1 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.

5.2.2 Semantics

We interpret the context Γ as usual:

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

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 effectul 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.

We now interpret the terms of the language 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

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

In order to explain the interpretation we work in the category C so the application of the monad T is explicit. We also remove the semantic brackets for the sake of removing clutter. Now, for λ-abstraction we work as follows. Assume a map $t : \Gamma \times A \to TB$. We have to define a map $\Gamma \to T(TB^A)$. By currying t we obtain $\Lambda t : \Gamma TB^A$ and by post-composing this map with η_{TB^A} we obtain the type $T(TB^A)$.

Function application is more tricky in that this is the point where we need the monad to be strong. For two maps $t_1 : \Gamma \to T(TB^A)$ and $t_2 : \Gamma \to TA$ We define the map $app(t_1, t_2) : \Gamma \to TB$ as follows:

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

References

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

[2] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'État, Université Paris VII, 1972.

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

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

[5] Alfio Martini. Category theory and the simply-typed lambda-calculus. 1996.

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

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

[8] Andrew M. Pitts. Polymorphism is set theoretic, constructively. In David H. Pitt, Axel Poigné, and David E. Rydeheard, editors, *Category Theory and Computer Science, Edinburgh, UK, September 7-9, 1987, Proceedings*, volume 283 of *Lecture Notes in Computer Science*, pages 12–39. Springer, 1987.

[9] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 513–523. North-Holland/IFIP, 1983.

[10] John C. Reynolds. Polymorphism is not set-theoretic. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 1984.

[11] Philip Wadler. Theorems for free! In Joseph E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359. ACM, 1989.