Finite Presheaf categories as a nice setting for doing generic programming.

Matias Menni

LIFIA, Departamento de informática, Universidad Nacional de la Plata.
C.C. 11, Correo Central,
1900, La Plata, Buenos Aires, República Argentina
E-mail: matias@sol.info.unlp.edu.ar
URL: http://www-lifia.info.unlp.edu.ar

Abstract

The purpose of this paper is to describe how some theorems about constructions in categories can be seen as a way of doing generic programming. No prior knowledge of category theory is required to understand the paper.

We explore the class of finite presheaf categories. Each of these categories can be seen as a type or universe of structures parameterized by a diagram (actually a finite category) C. Examples of these categories are: graphs, labeled graphs, finite automata and evolutive sets.

Limits and colimits are very general ways of combining objects in categories in such a way that a new object is built and satisfies a certain universal property. When concentrating on finite presheaf categories and interpreting them as types or structures, limits and colimits can be interpreted as very general operations on types. Theorems on the construction of limits and colimits in arbitrary categories will provide a generic implementation of these operations.

Also, finite presheaf categories are toposes. Because of this, each of these categories has an internal logic. We are going to show that some theorems about the truth of sentences of this logic can be interpreted as a way an implementing a generic theorem prover.

The paper discusses non trivial theorems and definitions from category and topos theory but the emphasis is put on their computational content and in what way they provide rich and abstract data structures and algorithms.

1 Preliminaries

This paper is about data structures and algorithms. Their peculiarity is that they are extracted from theorems from the branch of mathematics called Category Theory [10, 1, 5]. The paper is thought for readers with no knowledge of Category Theory.

Category Theory has been extensively applied to computer science [17, 12] (this last book has an extensive annotated bibliography). For example, the Constructive Algorithmics community [4, 11, 3] has used it as a vehicle for specifying recursive datatypes and deriving algorithms. Also realted to programming, monads (a categorical notion) have been used as a means to structure programs [16]. On the other hand, Category Theory has shown to be a very rich and powerful framework in which to unify several aspects of the semantics of programming languages [9, 6, 2] and references therein.

In this paper we describe how some constructions and theorems of category theory specialize to data structures and algorithms. First we observe that certain categories can be seen as types in the sense that their objects can be stored in a computer's memory. Examples of these categories or types are: graphs, labeled graphs, finite automata and evolutive sets. One advantage of using Category Theory to model these types is that they all can be described uniformly as finite presheaf categories.

Then we describe limits and colimits which can be seen as very general ways of combining objects. We show that in the case of finite presheaf categories, the construction of limits and colimits can be implemented. In these way we obtain algorithms that work generically with *any* finite presheaf category! A very clear example of the power of this approach is that the construction of the product of two graphs or finite automata or evolutive sets is performed by *one* generic program!

Also, finite presheaf categories are topoi. This means that they have (among other things) enough structure to interpret the usual conectives and quantifiers of first order logic. But this interpretation is very non standard and this gives rise to non standard logics with non standard truth values and a non standard notion of validity. In these paper we show an example of one of these logics and we argue that although we lack of clear examples, we believe that they could be used as a powerful programming tool. The evidence for this is that we can build a program to calculate the validity of a formula of the internal logic of any finite presheaf category.

In order to present these ideas in their full generality, non trivial notions from Category Theory must be introduced. We do this in sections 2 and 3. In section 2 there are also some examples close to computer science.

In sections 4 and 5, we describe how very generic algorithms are obtained as special cases of known theorems about limits in categories. We also present some examples in order to exemplify their genericity.

Section 6 steams from the fact that the category C-**Structures** (for every finite category C) is a topos [8, 15]. As such, it has an internal logic. The key observation is that this logic can be implemented and that the program obtained works for any category of C-structures. In this section, though, the categorical notions are not introduced as they are not needed to describe the algorithm.

2 What is a Category?

Definition 2.1 *A category C is given by the following data:*

a *class Obj(C) 'of objects'*

a class $Arr(C)$ 'of arrows among objects'

two functions dom,cod:$Arr(C) \to Obj(C)$ called 'domain' and 'codomain',
if $dom(f) = a$ and $cod(f) = b$ we write f:$a \to b$

a function $id_{(-)}$:$Obj(C) \to Arr(C)$ giving the 'identity arrow' for each object

a partial function $\circ$:$Arr(C) \times Arr(C) \to Arr(C)$ called 'composition'

These data satisfy the following axioms:

$dom(id_a) = cod(id_a) = a$

$f \circ g$ is defined if and only if $dom(f) = cod(g)$

If g:$a \to b$ and f:$b \to c$ then $f \circ g$:$a \to c$

If f:$a \to b$ then $id_b \circ f = f \circ id_a = f$

$f \circ (g \circ h) = (f \circ g) \circ h$ (in case compositions are defined)

■

The first example that comes to mind is the category of sets usually denoted by **Set**. Its objects are sets and its arrows are the functions among them. Usually, any mathematical structure together with the morphisms among them will form a category. For example, there exist categories of monoids and monoid morphisms, groups and group morphisms, rings and ring morphisms and in general for any algebraic structure. Another example (one that is going to be used a lot in this paper) is the category **FinSet**. Its objects are the finite sets and its arrows are the functions among them. Also, any set can be seen as a category. A set is just a category such that the only arrows are the identities.

We say that a category is *finite* if it has a finite number of arrows.

One curious fact about categories is that you can always turn around all the arrows and what you get is again a category. More precisely:

Definition 2.2 *Given a category C, we define C^{op} to be the category whose objects and arrows are those of C but with functions dom^{op}, cod^{op} and $\circ^{op}$ defined by:*
$dom^{op}(f) = cod(f)$, $cod^{op}(f) = dom(f)$ and $f \circ^{op} g = g \circ f$.

■

Note that, $(\mathrm{C}^{op})^{op} = \mathrm{C}$.

Arrows should not be assumed to be always functions that preserve some structure (in the examples above: the operations of the algebras). For example, any monoid can be seen as a category with just one object and the elements of the monoid as arrows. The composition of arrows is the monoid's operation and the identity arrow is the identity of the monoid.

Preorders are another source of examples. In fact any preorder can be seen as a category. The elements of the preorder are the objects of the category and there is an

arrow between a and b if and only if a $\leq$ b. Note two things: first, every object has an identity because the preorder is a reflexive relation and second, there is only one arrow between any two objects (actually, we could redefine a preorder as a category with this last condition).

We can draw some very small examples of categories.

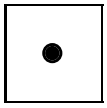

Figure 1: $\mathbf{1}^{op}$

Figure 1 is the category with just one point and its identity arrow (note that we are ommiting the identity arrow in the drawing) and figure 2 is the category with two points.

Figure 2: $\mathbf{2}^{op}$

For these two examples holds that the opposite category is the same as the original. That is $\mathbf{1}^{op} = \mathbf{1}$ and $\mathbf{2}^{op} = \mathbf{2}$.

Figure 3 is the category with two points and two parallel arrows between them.

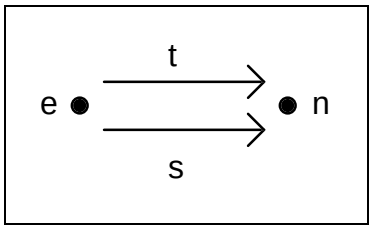

Figure 3: $\mathbf{G}^{op}$

Note that we have considered the previous categories as opposite categories. This may appear unnecessary but we do this in order to prepare the reader for the definition of presheaf and finite presheaf.

As a final example we introduce $\mathbf{0}$, the smallest category. It has no objects and hence no arrows.

In some way, the definition of a category says that every mathematical structure should always be presented together with a definition of morphism between two such structures. Because of this (and because we are going to use them) we define what is a morphism of categories.

Definition 2.3 *Let C and D be two categories, a functor F:C $\rightarrow$ D is a pair of functions F:Obj(C) $\rightarrow$ Obj(D) and F:Arr(C) $\rightarrow$ Arr(D) (note that we use the same letter for both functions) such that:*

Ff:Fa $\to$ *Fb for every arrow f:a* $\to$ *b in C.*

$Fid_a = id_{Fa}$ *for every object a in C.*

$F(f \circ g) = (Ff) \circ (Fg)$ *if f and g can compose.*

■

Let us review some examples.

Of course, for every category C, there is an identity functor id_C which sends every object and arrow to itself.

Among categories of algebraic structures there are several 'inclusions'. For example, there exists a functor U from the category of groups to the category of monoids such that sends every group to its underlying monoid and every group morphism to the same morphism (which is obviously a morphism between the underlying monoids of its domain and codomain).

There is just one functor from the category **0** to any other category, the empty functor. Dually, there is just one functor from any category to **1**: it assigns every object to the one object in **1** and every arrow to the unique arrow in **1**.

For every category J and object c in a category C we can define the constant functor as follows:

Δ_c:J $\to$ C

Δ_ca = c for every object a in J

Δ_c(f:a $\to$ b) = id_c for every f in J

Also, functors can be seen as *structures* or data types.

2.1 C-structures

To see how functors can be seen as structures, let us have first an intuitive discussion. Let $\mathbb{N}$ be the set of natural numbers and for any n $\in \mathbb{N}$, we let [n] be the set {1,...,n}. A function f:[n] $\to$ A can be seen as a list of n elements from A; let us say, an [n]-collection. Also f:$\mathbb{N}$ $\to$ A can be seen as an infinite list of elements in A, that is, a $\mathbb{N}$-collection. Actually, we could see any function f:S $\to$ A from a set S as a S-collection.

Now replace the set S for some finite category C, the set A for the category **FinSet** and imagine a functor F:C^{op} $\to$ **FinSet**. Such a functor can be seen as selecting a finite set for each object of C, but it also can be seen as selecting a function for each arrow in C^{op}. Besides, the selection of functions must be done in such a way that the selection of sets respect the domain and codomain of the functions. Moreover, compostion must be preserved. So it would not be a good idea to name such a functor a C-collection. Let us call it a C-structure.

Let us look at some examples.

First note that a finite set A is just a functor A:**1** $\to$ **FinSet**. So we can see **1**-structures as the type of finite sets.

We can try to aim a little higher and define the type of pairs of finite sets. Its elements are the **2**-structures.

We can get ambicious and try to define the type of finite graphs. A finite graph is just a finite set of nodes and a finite set of edges such that each edge has associated two nodes, its source and target. That is, a finite graph F consists a finite set Fn of nodes, a finite set Fe of edges and two functions Fs,Ft:Fe $\rightarrow$ Fn. But this is just a **G**-structure F!

Let us build new types over this last example. What if we wanted labeled graphs? Then we should 'add' a set of labels and 'say' for each edge how it is labeled. Then a finite labeled graph is a **LG**-structure (see figure 4).

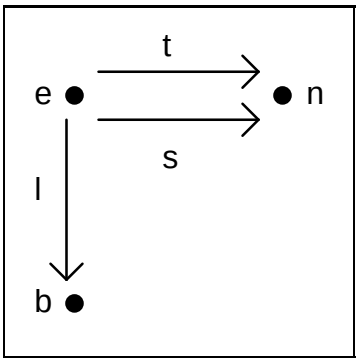

Figure 4: $\mathbf{LG}^{op}$

Consider now finite automata. We could implement them as finite labeled graphs with a distinguished 'initial' state and some distinguished 'final' states. Given a labeled graph F we could select a subset of final (or 'accept') states with a function Ff:Fa $\rightarrow$ Fn from some set Fa to the set of nodes of the graph. We could also select an initial (or 'start') state with another function Fi:Fc $\rightarrow$ Fn. Such a structure is a **FA**-structure (see figure 5).

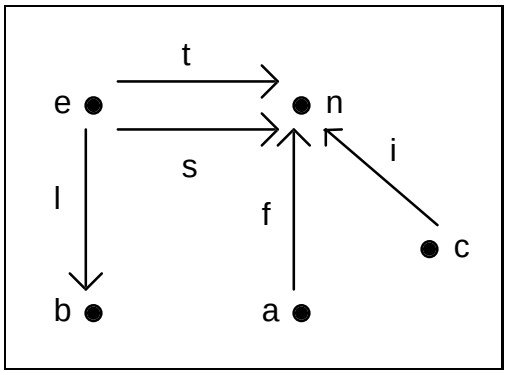

Figure 5: $\mathbf{FA}^{op}$

Note that there are some **FA**-structures that are not really automata. For example, an **FA**-structure F with Fs empty would be interpreted as a finite automata without initial state. On the other hand, there could be more than one node in the image of Fi, so F would have more than one initial states.

We dismiss this facts as unimportant for us, as it is a usual situation in the activity of programming that programmers should be careful in defining instances of some datatype.

It is worth noting that these examples generalize easily if we put **Set** instead of **FinSet** and drop the assumption that C is finite. A functor F:$C^{op} \to$ **Set** is called a *Presheaf* [14, 15]. That is why a functor F:$C^{op} \to$ **FinSet** for a finite C is called a finite presheaf. Because of the above examples and because we want to see them as data structures we rather call them C-structures.

In this paper we are going to concentrate on C-structures instead of presheaves. The main reason for doing this is that C-structures can be implemented.

Although it is not clear what an elegant implementation of C-structures would be, it is clear that there is no computational problem in storing all the information that any C-structure describes. Actually, it is only a selection of a finite number of finite sets and a finite number of total functions among them.

2.2 The category of C-structures

As we said before, every notion of structure should come together with a definition of what is a morphism between any two of them.

Definition 2.4 *Let F,G:C $\to$ D be two functors. A natural transformation α:F $\to$ G is a family of arrows α_a:Fa $\to$ Ga one for each object a of C, such that for every arrow f:a $\to$ b in C, $\alpha_b \circ Ff = Gf \circ \alpha_a$.* ■

It is easy to check that there exists an identity natural transfomation for each functor. It is also easy to define the composition of two natural transformations and to check that it is associative.

In this way, any two categories C and D give rise to a functor category D^C. Its objects are functors from C to D and its arrows are natural transfomations. In particular, for any finite C we have the category **FinSet**$^{C^{op}}$ which we rather call C-**Structures**.

Let us consider the case of a natural transformation between two **G**-structures (finite graphs).

Let F and G be two **G**-structures. A natural transformation α:F $\to$ G is a pair of functions α_n:Fn $\to$ Gn and α_e:Fe $\to$ Ge (one from nodes to nodes and one from edges to edges) such that:

$$\alpha_n \circ Fs = Gs \circ \alpha_e \quad and \quad \alpha_n \circ Ft = Gt \circ \alpha_e$$

If we replace Fs and Gs by source and Ft and Gt by target we obtain:

$$\alpha_n \circ source = source \circ \alpha_e \quad and \quad \alpha_n \circ target = target \circ \alpha_e$$

which is just the definition of a morphism α of graphs!

It is very easy to check that natural transformations of **1**,**2**,**LG** and **FA**-structures are respectively functions, pairs of functions, morphisms of labeled graphs, and 'morphisms of finite automata' (be careful with the peculiar cases discussed above).

Note also that in this setting of finite sets and total functions, we can write a program to test if a set of functions is a natural transformation.

It is important to stress how the language of category theory has helped to define a notion of morphism of structures that is independent of any particular structure. The definition of natural transformation relies only on an arbitrary category C.

We shall see that something similar happens with constructions in functor categories. Because of this, we will be able to obtain generic algorithms in the sence that they are parameterized by a finite category.

3 Limits and colimits

In this section we introduce limits and colimits. These are very general ways of combining or merging objects in a category in order to obtain a new object that satisfies certain special property.

First we introduce some particular examples of limits among sets. You should bear in mind though, that we plan to discuss the implementation of these ideas. Because of this, it may be useful to think that we are working only with finite sets.

3.1 Terminal objects, products and equalizers in Set

Consider the singleton set $1 = \{*\}$. It has a very interesting property.

Proposition 3.1 *For every set b, there exists a unique function from b to 1. It assigns * to every object of b.* ∎

Because of this property, any singleton set is called *terminal*. The property is called the *universal property of the terminal object*.

Now, given two sets a and b, we can build their cartesian product:

$$a \times b = \{(x,y) \mid x \in a \text{ and } y \in b\}$$

We usually define the product of two sets together with two functions $\pi_1 : a \times b \to a$ and $\pi_2 : a \times b \to b$ (the projections). The product and these two functions satisfy the following property, usually called *universal property of the product*.

Proposition 3.2 *For every set c and two functions $f : c \to a$ and $g : c \to b$ there exists a unique function $(f,g) : c \to a \times b$ such that $\pi_1 \circ (f,g) = f$ and $\pi_2 \circ (f,g) = g$. Explicitely, $(f,g)(x) = (f(x),g(x))$.* ∎

Think of this universal property as saying that we have built the product in *the right way*.

Consider again two sets a and b. Consider also two functions $f,g : a \to b$. We can build the set $E = \{x \in a \mid f(x) = g(x)\}$ and a function $e : Eq \to a$ that takes every $x \in Eq$ to its copy in a. This set Eq together with the function e are called the equalizer of f and g. They satisfy the following *universal property of the equalizer*.

Proposition 3.3 *For every set c together with a function $h : c \to a$ such that $f \circ h = g \circ h$ there exists a unique function $j : c \to Eq$ such that $e \circ j = h$.* ∎

Again, think of this property as saying that this is *the right way* to find the part of a for which f and g are equal.

Let us try to generalize this constructions in order to find out what is it that they have in common. At first, it may appear that we are trying to make something very strange out of something very simple. But in future sections we are going to see the benefits.

First we said that we were considering two sets a and b. We already know that this is the same as considering a functor P:$\mathbf{2} \to \mathbf{Set}$ with P1 = a and P2 = b. Then we built their product a$\times$b. We also know that we can consider this set as the constant functor $\Delta_{a\times b}$:$\mathbf{2} \to \mathbf{Set}$.

We now have all the objects in the definition of product presented as functors. The two prejections can be presented as a natural transformation π:$\Delta_{a\times b} \to$ P. As we are considering functors from **2** then such a natural transfomation is a pair of functions π_1:$\Delta_{a\times b}1 \to$ P1 and π_2:$\Delta_{a\times b}1 \to$ P2. This is just a pair of functions π_1:a$\times$b $\to$ a and π_2:a$\times$b $\to$ b.

Note that in the universal property of the product we considered other set c together with its two 'projections'. This is just another natural transformation π':$\Delta_c \to$ P.

We can call P a **2**-*diagram* and for any c we are going to call any natural transformation α:$\Delta_c \to$ P a *cone for* P.

Also for any to cones α:$\Delta_c \to$ P and β:$\Delta_d \to$ P, an arrow f:$\alpha \to \beta$ is an arrow f:c $\to$ d such that $\beta_1 \circ$ f = α_1 and $\beta_2 \circ$ f = α_2.

We can now formulate the universal property of the product in this language.

Let P be a **2**-diagram such that P1 = a and P2 = b. Then, for any cone α for P there exists a unique f:$\alpha \to \pi$ to the cone π:$\Delta_{a\times b} \to$ P.

We could do something similar for the terminal object and equalizers, we should only consider **0** and **G**-diagrams respectively instead of **2**-diagrams. Actually, we could do something similar for J-diagrams where J is any category!

Definition 3.4 *Let J be a category. A J-diagram in a category C is a functor D:J $\to$ C.* ■

Definition 3.5 *Let D be a J-diagram in C. A cone for D is an object a in C together with a natural transformation α:$\Delta_a \to$ D.* ■

Definition 3.6 *Given D a J-diagram in C and two cones α:$\Delta_a \to$ D and β:$\Delta_b \to$ D, an arrow f:$\alpha \to \beta$ is an arrow f:a $\to$ b such that for all i in J, beta$_i \circ$ f = α_i.* ■

Definition 3.7 *A limit for a J-diagram D in C is a cone α:$\Delta_l \to$ D such that for any other cone β there exists a unique arrow f:$\beta \to \alpha$. The cone α is called limiting cone.* ■

A fundamental property of limits is that they are *almost* unique.

Two objects l and l' are *isomorphic* if there exist arrows i:l $\to$ l' and i':l' $\to$ l such that i $\circ$ i' = id$_{l'}$ and i' $\circ$ i = id$_l$. With this definition we can formulate:

Proposition 3.8 *(Uniqueness up to isomorphism) Let D be J-diagram in C. If there exist two limits α:$\Delta_l \to D$ and α':$\Delta_{l'} \to D$, then l and l' are isomorphic.* ∎

Intuitively this property means that for any purpose any one of the two objects would do as well as the other. Just as in **Set**.

Think of a limit in the category **Set** or **FinSet**. Intuitively, the limit for a J-diagram is a subset l of the product of the sets in the diagram together with its projections. This subset l is the biggest one that *fits the diagram*. It *fits the diagram* in the following sense. Imagine there is function f:a $\to$ b in the diagram. There are pojections p_a:l $\to$ a and p_b:l $\to$ b which are part of the limiting cone. Then if t $\in$ l it holds that f $\circ$ p_a(t) = p_b(t).

Limits are used to define terminal objects, products and equalizers in any category. These are limits for **0**, **2** and **G**-diagrams respectively. Note that this can be done because the definition of limit relies not in its representation but in the universal property of the limiting cone. Uniqueness up to isomorphism says that it does *not* matter that the definition does not give a particular representation.

3.2 Initial object, coproducts and coequalizers in Set

A colimit in C is just a limit in C^{op}. This definition will probably not give an intuitive idea of what colimits are. So let us look at some examples in **Set**.

Proposition 3.9 *For any set a there exists a unique function from $\emptyset$ to a; the empty function.* ∎

Because of this $\emptyset$ is called the *initial object*. The property is called the *universal property of the initial object*.

Let us consider again two sets a and b and build their sum:

$$\text{a+b} = \{(0,\text{x}) \mid \text{x} \in \text{a}\} \cup \{(1,\text{y}) \mid \text{y} \in \text{b}\}$$

Again we have two functions in_1:a $\to$ a+b and in_2:b $\to$ a+b. Moreover, this construction also satisfies a universal property: the *universal property of the coproduct*.

Proposition 3.10 *For every set c and two functions f:a $\to$ c and g:b $\to$ c there exists a unique function [f, g]:a+b $\to$ c such that [f, g] $\circ$ in_1 = f and [f, g] $\circ$ in_2 = g. Explicitely,*

$$[f,g](0,x) = f(x) \quad \textit{and} \quad [f,g](1,y) = g(y)$$

∎

If you look carefully at the definition of sum and its universal property you will find out that it is just the definition of product and its property but with the arrows (functions) turned around. That is why we call this construction *coproduct*.

Now, given two sets a and b and two functions f,g:a $\to$ b. We can build the set coEq = b/$\sim$ where $\sim$ is the least equivalence relation on b which contains all pairs (fx,gx) for x $\in$ a. We also have a function ce:b $\to$ coEq which sends every element on b to its equivalence class. Together they satisfy the following *universal property of the coequalizer*.

Proposition 3.11 *For every set c together with a function h:b $\to$ c such that h $\circ$ f = h $\circ$ g there exists a unique function j:coEq $\to$ c such that j $\circ$ ce = h.* ■

Note again that this is a just the 'turned around' version of the universal property of the equalizer.

We could actually turn around the definition of a limit.

Definition 3.12 *Let D be a J-diagram in C. A cocone for D is an object a in C together with a natural transformation α:D $\to$ Δ_a.* ■

Definition 3.13 *Given D a J-diagram in C and two cocones α:D $\to$ Δ_a and β:D $\to$ Δ_b, an arrow f:α $\to$ β is an arrow f:a $\to$ b such that for all i in J, beta$_i$ = f $\circ$ α_i.* ■

Definition 3.14 *A colimit for a J-diagram D in C is a cocone α:D $\to$ Δ_l such that for any other cocone β there exists a unique arrow f:α $\to$ β.* ■

Colimits are also unique up to isomorphism and they are used to define initial objects, coproducts and coequalizers in arbitrary categories. The intuition behind colimts is similar to that of limits. Just replace products with coproducts and projections by inyections.

4 The implementation of limits and colimits

When we need to study the properties of limits in general sometimes it is very good not to be attached to a particular representation. In this way we can abstract from unecesary details. On the other hand, when we need to store something in a computer memory, we can not do without a representation. Moreover, we need a *finite* representation.

In this section we are going to describe how to build arbitrary finite limits and colimits in categories. We shall pay special atention to the case of limits in **FinSet**.

We have defined limits and colimits for arbitrary diagrams and arbitrary categories. Yet we only know how to build **0**,**2** and **G**-diagrams in **Set**. Surprisingly, it is enough to know how to calculate these ones in order to calculte the limit or colimit of any diagram in **Set**.

Before starting the construction of limits note that if a category has binary products then it has products of any finite arity: just iterate the binary product. We denote these products with

$$\prod_i$$

for some i varying over the elements of some finite set I (note that they satisfy a universal property that is very similar to that of the binary product, it just has more projections). By convention the iterated product over the empty set is the terminal object.

Theorem 4.1 *If a category C has a final object, equalizers and binary products then C has all finite limits.*

Proof. Let J be a finite category and F a J-diagram in C. We can build

$$R := \prod_{i \ in \ J} Fi \quad and \quad S := \prod_{u:j \longrightarrow k} Fk$$

R is the product of the F-images of objects in J and S is the product of the F-images of the codomains of arrows in J.

For each component Fcod(u) of the product S we have a projection from the product R. Because of the universal property of product S, there exists a unique arrow f:R $\rightarrow$ S which relates these two cones.

Also, for each arrow u in J we have a projection $p_{dom(u)}$:R $\rightarrow$ Fdom(u) which we can compose with Fu obtaining thus for each arrow u in J a projection

$$(Fu \circ p_{dom(u)}):R \rightarrow Fcod(u)$$

Again, by the universal property of the product there exists a unique g:R $\rightarrow$ S which is an arrow between the cones. So we have two arrows f,g:R $\rightarrow$ S.

Because of the assumption, there exists an equalizer e:Eq $\rightarrow$ R of this two arrows. It can be proved that the family of arrows $\mu_i = p_i \circ e:Eq \rightarrow Fi$ form a cone which is a limit of the J-diagram F.

■

Note that if we just consider finite sets and total functions, then we can write programs to build products, equalizers, coproducts and coequalizers. The concrete representations we have given in the previous section give rise to very simple algorithms (the building of coequalizers is not trivial but it can be done). We also know that **FinSet** has a terminal object. So the above construction is just a program that builds the limit for any given J-diagram F in **FinSet** for any finite J.

It is very important to note that if we replace products by coproducts and equalizers by coequalizers we obtain the construction for an arbitrary colimit. Also this construction specializes to an algorithm when we consider the category **FinSet**.

5 Limits and colimits among C-structures.

Now that we understand the concept of limit and colimit we may ask if they exist among C-structures. Note that this is not a trivial question. The definition of limits makes sence in any category, particularly in any functor category such as a category of C-structures. Yet we only know how to build limits and colimits in **FinSet**. Besides, there is another related question which is of particular interest to us. Assuming limits and/or colimits exist, can we write a program to build them?

For any functor category X^J and object j in J there exists a functor 'evaluation' defined as follows:

E_j:$X^J \rightarrow$ X

E_jH = Hj for any functor H:C $\rightarrow$ X

$E_j(\alpha$:H $\rightarrow$ H') $= \alpha_j$:Hj $\rightarrow$ H'j for any natural transformation α

This functor evaluates its argument at a given point. Using this functor we can formulate the following theorem. The proof can be found in [10]. We are not going to present it here as an algorithm can be extracted just from the satement of the theorem.

Theorem 5.1 *For all c in C, and S in* X^C,
$E_c(Limit(S)) = Limit(E_cS)$ *and*
$E_c(CoLimit(S)) = CoLimit(E_cS)$. ∎

That is, in any functor category, limits and colimits can be calculated pointwise (provided pointwise limits exist).

When we replace X by any category C-**Structures**, and assume that J is finite the theorem is just the description of an algorithm that calculates the limit or colimit of any given J-diagram of C-structures.

Let us look at some examples.

First, consider the product of two **G**-structures (i.e. finite graphs). The product defined as a limit is just the usual definition of product of graphs. This is one of the most simple ways of combining graphs. Suppose now that the two graphs F and G represent roads and cities or machines and connections (or something else). Suppose you have obtained these grpahs from different sources (e.g. different tourism offices or departments in an enterprise) and that you have to make one graph out of the ones you have. Obtaining their disjoint union will not work as their may be nodes or edges in the two graphs representing the same thing in the real world. What we should obtain is a merge of the two graphs where the nodes that represent the same things are collapsed. Clearly we can represent the nodes and edges that represent the same things by another graph H. This graph H can be embbeded in the other graphs by two 'inclusions' i:H $\rightarrow$ F and i':H $\rightarrow$ G (note that they are not strict inclusions as F and G may use different representations for the same entity in the real world). These inclusions give rise to a diagram. To obtain the desired 'mixed' graph you just ask the computer to calculate the colimit.

Consider two finite automata F and G (i.e. two **FA**-structures). We can ask the computer to calculate their product using the algorithm given by the theorem in the previous section. The result is almost the construction given in [7] to prove that the class of type 3 languages is closed under intersection. Explicitely, a new automaton which set of states is the product of the sets of states of F and G, transitions are pairs of transitions, the final states are pairs of final states and the initial state is the pair with components the initial states of F and G. The main difference with the construction in [7] is that their new automaton is labeled with the original alphabet. On the other hand our new automaton is labeled with pairs of elements of the original alphabet. Yet if for each symbol x that our automaton is supposed to read we replace it by (x,x) then the language that it recognizes is the same.

It is also known that type 3 languages are closed under union. A first attempt to prove this is to build the disjoint union of any two automata F and G. Yet this attempt fails as it is not clear what should be the initial state of the new 'automaton'. To solve this, Hopcroft and Ullman add a new state. We are going to solve this by building the right

colimit. This is the colimit of the following diagram. First put the two automata F and G. Add the automaton A with only one state which is initial and no transitions or final states. Also add two arrows f:A $\to$ F and g:A $\to$ G. These arrows assign the unique state of A to the initial states of F and G respectively. In this way the program that builds the colimit will make the disjoint union of the three automata and then obtain the quotient in which the initial states have collapsed to one state.

More C-structures and more complicated 'merging conditions' give rise to more complicated diagrams. Yet all (co)limits would be calculated by the same algorithm. In some sence (similar to that of natural transformations) these algorithms are 'generic' as they work for any category of C-structures!

One area of possible application of this '(co)limit programming' is the implementation of GIS. These systems work with different layers of information and they have to 'merge' this layers in order to respond to queries. Limits and colimits seem to be a very natural way solving these problems.

6 The internal logic of C-structures and its use.

For every finite C, C-**Structures** is a *topos*. Because of this, the usual logical operators and quantifiers can be interpreted in these categories. It turns out that in the case of C-structures, this logic can be implemented. In this way we have a kind of 'first order logic' which we can use as a query language for C-structures. This logic behaves differently from the usual first order logic as it takes into account the structure of the category C. Because of this, we have a different logic for every C, yet a theorem about the validity of the formulae of these logics will provide a sort of 'theorem prover' that is independent of the category C we are considering.

In this section we introduce the language of these logics and the theorem from which we extract the theorem prover. We also present some toy examples in order to show how it could be used and how it behaves according to the structure of C.

In this section we are going to work with a fixed finite category C and the category C-**Structures**. We are assuming that we have variables X, Y, Z... representing sorts and variables x, y, z... of that sort. Also, for each pair X, Y of sorts we assume there is another sort (X $\longrightarrow$ Y) and variables f, g ... of that sort.

With this elements we are going to define a typed language and we are going to present a computational interpretation.

We define terms and formulae as follows.

1. any variable of sort X is a term of sort X
2. if t is of sort X and f is a variable of sort (X $\to$ Y) then (ft) is a term of sort Y
3. if t and t' are terms of sort X then (t = t') is a *formula*
4. if x is a variable and t and t' are *formulae* then so are (t$\wedge$t'), (t$\vee$t'), (t$\Rightarrow$t'), ($\neg$t), ($\forall$x)t and ($\exists$x)t

We can associate a C-structure X to each sort X, and a natural transfomation f:X $\to$ Y to each variable f of sort (X $\longrightarrow$ Y). In what follows when we speak of a formula ϕ we are going to assume that we have already associated C-structures and natural transfomations to sorts and variables f in ϕ.

There exists a notion of validity for the formulae of this language. So it makes sence to ask if a formula ϕ is c-*valid* (written c$\models\phi$) for some assignment to the free variables in ϕ. We are not going to present the definition here because we would need a lot more topos theory than what fits in this paper. Yet, thanks to the theorem below, we do not need it in order to calculate it.

We use $\overline{x}$ to denote a list of variables x_1,...,x_n such that x_i has sort X_i. A c-assignment $\overline{\alpha}$ to $\overline{x}$ is a list of elements α_1,...,α_n such that $\alpha_i \in X_i$c. Finally if f:c $\to$ c' we use f$\overline{\alpha}$ to denote a the list of elements $X_1f(\alpha_1)$,...,$X_nf(\alpha_n)$.

It must be noted that when we assign values to the free vars of a formula then all terms in it can be evaluated to an element of a set.

Theorem 6.1 *Let $\phi(\overline{x})$ and $\rho(\overline{x})$ br formulae with free variables in $\overline{x}$ and θ a formula which may also have y of sort Y as a free variable. Then for any c-assingment $\overline{\alpha}$ to $\overline{x}$ we have:*

c$\models(\alpha_i = \beta_j)$ iff α_i equals β_j
c$\models(\phi(\overline{\alpha})\wedge\rho(\overline{\alpha}))$ iff c$\models\phi(\overline{\alpha})$ and c$\models\rho(\overline{\alpha})$
c$\models(\phi(\overline{\alpha})\vee\rho(\overline{\alpha}))$ iff c$\models\phi(\overline{\alpha})$ or c$\models\rho(\overline{\alpha})$
c$\models(\phi(\overline{\alpha})\Rightarrow\rho(\overline{\alpha}))$ iff c$\models\phi(\overline{\alpha})$ implies c$\models\rho(\overline{\alpha})$
c$\models(\neg\phi(\overline{\alpha}))$ iff for no f:c $\to$ d in C^{op} c$\models\phi(f(\overline{\alpha}))$
c$\models(\forall y)\theta(\overline{\alpha},y)$ iff for all f:c $\to$ d in C^{op} and all $\beta\in Y(d)$, one has d$\models\theta(f(\overline{\alpha}),\beta)$
c$\models(\exists y)\theta(\overline{\alpha},y)$ iff for there exists a $\beta\in Y(d)$ such that c$\models\theta(\overline{\alpha},\beta)$

■

It is important to say that the formulation of this theorem gives rise to an algorithm because we are considering C-structures. Note that if we were considering presheaves then the clauses for $\forall$ and $\exists$ would involve a potentially infinite set Y(d). Also, if C was not finite then clauses for $\neg$ and $\forall$ would be dealing with an infinite number of arrows.

Let us analyze some examples. First, consider the formula (x = x') with x and x' variables of sort X.

For any pair of elements α,α' $\in$ Xc we can ask the computer to calculate if c$\models(\alpha = \alpha')$. It is important to think the formula (x = x') as a generic program that works for *any* category of C-structures.

Now consider the formula $(\exists x)(ix = z) \vee (\exists y)(jy = z)$ where x,y,z are of sorts X,Y and Z respectively and i:X $\to$ Z and j:Y $\to$ Z.

For any c in C we could select an object α from Zc and ask the computer to calculate if there exists an element in Xc or Yc such that it is mapped via i or j into α. In some sence we have written a program that we can use to test if any two given C-structures 'cover' a third.

Let us look at a little more complicated example:

$$(\forall x)((\exists y)((\exists z)((ix = z) \Rightarrow (jy = z))))$$

Assume that we are working with the category **G-Structures**. Consider X and Y to be subgraphs of Z and i,j to be the respective inclusions. It turns out that the formula is n-valid if and only if all the nodes of X are also nodes of Y. It is e-valid if and only if X is also a subgraph of Y. The reader is encouraged to calculate these facts using the theorem above.

Intuitively, when we ask if a formula is c-valid, we are asking something about the sets that the C-structures associated with variables assign to c. If the formula is built without $\neg$ and $\forall$ then the logic behaves classicaly. You can solve the base cases which are just equalities and then use the classical truth tables to resolve the inductive cases. On the other hand if $\neg$ or $\forall$ are used in the term then the structure of C starts to show its influence.

Assume we are working with **G**-structures and consider the formula

$$\phi(\mathrm{y}) = (\exists x)(ix = y)$$

Suppose we have assigned to sort Y the graph with three nodes 1,2 and 3 and an edge a from 1 to 2. Suppose also that X was assigned the subgraph with nodes 1 and 2 and edge a and i was assigned the inclusion.

Let us calculate if e$\models(\forall y)((\exists x)(ix = y))$ holds. We must consider three cases.

1. id:e $\to$ e
2. s:e $\to$ n
3. t:e $\to$ n

Case 1. For all $\beta \in$ Ye we should calculate if e$\models(\exists x)(ix = \beta)$. This is easy as a is the only arrow in graph Y and a is also in X.

Case 2. For all $\beta \in$ Yn we should calculate n$\models(\exists x)(ix = \beta)$. But this fails when we consider $\beta = 3$.

So e$\not\models(\forall y)(\phi(y))$. At first this may appear contraintuitive as all arrows in Y are also in X. One way to look at this is to think that the arrows in C^{op} describe what 'parts' of the C-structures are related and that the quantifier $\forall$ takes this into account.

So c$\models(\forall x)(...)$ is *not* asking wether 'for all things in Xc' but 'for all the things in the parts of X that are realted to part c'.

Now consider the formula $\neg\phi$(y) with the same graph assigned to Y and the graph with nodes 1 and 2 but no edges to X. Again, i was assigned the inclusion. Let us calculate if e$\models\neg(\exists x)(ix = a)$ holds. It must be the case that

1. e$\not\models(\exists x)(ix = id(a))$
2. e$\not\models(\exists x)(ix = s(a))$
3. e$\not\models(\exists x)(ix = t(a))$

The first one holds (meaning, the formula is not e-valid) as there are no arrows in X. But n$\models(\exists x)(ix = 1)$ and n$\models(\exists x)(ix = 2)$ hold. So in spite of the fact that there are no edges in X, e$\models\neg(\exists x)(ix = a)$ does not hold!

This is a little harder to explain. It is true that edge a is not in X yet it is 'almost' there as both its source and target are. We could agree that if its source was not in X then edge a would be 'less' there.

In fact, e$\models\neg\phi$(a) would hold if none the source or target were in X.

We have presented a couple of toy examples in order to show how the internal logic of C-structures could be effectively used as a query language. The full power and utility of this notion of validity is still to be explored.

Again, we find that this theorem can be easily programmed. In this way we can look at $\models$ as if it were a very generic program for it would work for any category of C-structures.

Finally we should stress that the language presented here is a very limited one. There are more powerful versions. Also, the internal language of a topos is usually presented slightly differently, we have chosen this way because it leads easier to an implementation.

7 Future work.

One obvious line of future work is the developement of a concrete implementation of these ideas. As we said, it is clear that there is no computational problem for doing this, but it would be interesting to obtain an implementation that would let the programmer work as if he was doing category theory. Such implementation would be excellent for experimenting with these ideas an find more important examples and new areas of application.

It would be very interesting to go on exploring what and how theorems specialize to algorithms when applied to C-structures (such as adding modal operators [13]). Also we should study other classes of categories such that theorems specialize to algorithms when applied to them.

8 Acknowledgements

I would like to thank Drs. Gonzalo Reyes, Marta Sagastume and Adriana Galli for lecturing and organizing the course on Presheaf Categories at the Mathematics Department of the Universidad Nacional de La Plata. That course has been the main motivation for writting this paper. I would also like to thank Lic. Baum and Dr. Rossi for their particular way of stimulating the writting of this paper and Lic. Martinez Lopez for helping with some technical problems.

References

[1] *Categories, Types and Structures: An introduction to Category Theory for the Working Computer Scientist* Asperti y Longo. The MIT Press. 1991.

[2] *Axiomatic Domain Theory in Categories of Partial Maps* M. Fiore. Cambridge University Press,1996.

[3] *Law and order in Algorithmics* M. M. Fokkinga. Ph.D. Thesis, Technical University of Twente, The Netherlands. 1992. Available from URL `http://hydra.cs.utwente.nl/ fokkinga/mmfphd.html`.

[4] *A typed lambda calculus with categorical type constructors* Ph.D. thesis, University of Edinburgh, Dept. of Computer Science, 1987.

[5] *Category Theory* H. Herrrlich y G. E. Strecker. Allyn and Bacon. 1973.

[6] *First steps in Synthetic Domain Theory* J. M. E. Hyland. Category Theory, volume 1488 of Lecture Notes in Mathematics. Springer Verlag, 1991.

[7] *Introduction to automata theory, languages and computation* J. E. Hopcroft and J. D. Ullman. Addison Wesley.

[8] *Topos Theory* P. T. Johnstone. Academic Press. 1977.

[9] *Introduction to Higher Order Categorical Logic* J. Lambek y P. Scott. Number 7 in Cambridge Studies in Advanced Mathematics. Cambridge University Press. 1988.

[10] *Categories for the working mathematician* S. Mac Lane. Springer Verlag. 1971.

[11] *Data structures and program transformation* G. R. Malcolm. Sience of Computer Programming. 1990.

[12] *Basic Category Theory for computer scientists* Foundations of Computing Series. MIT Press.

[13] *A topos theoretic approach to reference and modality* G. Reyes. Notre Dame Journal for Formal Logic, 32. 1991

[14] Notes from the course *Presheaves* lectured by Dr. Gonzalo Reyes and Dr. Marta Sagastume at the Mathematics dept. (UNLP) during 1996.

[15] *Sheaves in geometry and logic: a first introduction to topos theory* S. Mac Lane and I. Moerdijk. Springer-Verlag, Berlin. 1992.

[16] *The essence of functional programming* P. Wadler. Proceedings of the Nineteenth ACM symposium on Principles of Programming Languages. 1992

[17] *Categories and Computer Science* R. F. C. Walters. Volume 28 of Cambridge Computer Science Texts. Cambridge University Press.