Programming Language Syntax: Expressing Sharing

Isabel L. Cafezeiro
Departamento de Ciência da Computação
Universidade Federal Fluminense
Niterói, RJ - Brasil
e-mail:isabel@dcc.ic.uff.br

Edward Hermann Haeusler
e-mail:heramnn@inf.puc-rio.br

PUC-RioInfo.MCC 09/99, May,1999

Abstract: This text presentes an operator to syntactically express sharing. The operator will be defined very abstractly, not considering a particular application, and will deal with collection of entities instead of single entities. The text also proposes further developments in the semantic level.

Keywords: Syntax of Programming Language, Formal Semantics, Programming Language, Theory of Computation, Categories.

Resumo: Este texto apresenta um operador para expressar compartilhamento sintaticamente. O operador será definido abstratamente, sem considerar uma aplicação particular, e deverá ser aplicado a coleções de entidades, ao invés de entidades simples. O texto também indica direções para trabalhos futuros no nível semântico.

Palavras-chave: Sintaxe de Linguagem de Programação, Semântica Formal, Linguagem de Programação, Teoria da Computação, Categorias.

Contents

1 Introduction

To talk about things that are put together one can use the product operator. Such operator (to simplify, the *binary* product) provides a way of thinking about two separate entities as a single one. The operation is reversible: the first and second projections can break the product restoring the individuality of each entity. The reversion is possible because, when put together, the entities do not interact.

But if the question is to put two entities together in a way that a component of one is fused with a component of the other, then the product operator does not fit the purpose any more. Furthermore, there is not a 'natural' way of defining the projections because part of one entity becomes part of the other.

This situation seems to be very tricky but, in fact, it happens all the time in computer science. For instance, two programs can run in parallel sharing resources; in syntax analysis, two productions can be applied in parallel, having both left sides matching the same symbols.

Wondering about this operation, we promptly realise that something has to be known about the composition of each entity, as the component to be fused is internal.

A first way of dealing with this question is to define the operation considering the kind of component to share. This approach is supported by the fact that things to be shared must have something in common. Since entities that have a common part must be treated together, it makes sense to define one operation to each of these groups. We have, then, several groups, say, *categories*, in which the sharing can be performed. The next step is to provide ways of passing from one group to another in other to make all the entities to interact.

A second way would be to adopt a level of abstraction by parameterizing, that is, defining one operation where the information about the entities' internal components is another operand.

The strong reason for which we prefer the first approach to the second is that it makes the definitions categorised, using the informal meaning of this word. That is, one can have the vision of the whole collection of groups, but one can also perform a kind of *zoom*, and see how entities of a particular group interact themselves. The weak reason is that parameterizing would make the operator definition awkward, as operands would have very distinct nature, one being part of the other.

The above remark is about the *sharing* itself, or how the *sharing* information should be presented in order to perform the operation. Prior to that point is the way that the entities must be structured to make the operation possible. Concerning this, three questions must be raised:

- *what* is to be shared?
- *where* do these things reside?
- *how* connect the *what* and the *where*?

The *what* question characterises the elements that will be fused. The operation considers that the component to be shared belongs to a fixed class of components. The idea is that we can vary the size, but not the type. As an example, if *memory* is to be shared, then one can share as many components as necessary, as long as they are *memory positions*.

The *where* question characterises the entities from which these elements are components. As we are focusing in the components to be shared, the entities must be described with respect to these components. So, we put in the same group entities that can be built with the same number of components. Again, if *memory* is to be shared and we are considering programs, commands etc, then programs, commands etc, that need n *memory positions* to be built, are grouped together.

Finally, the *how* question is about the connection of both, sharing and entities' descriptions. The point here is obvious: in a group of entities that are built with n components, *no more than* n components can be shared.

Let us remark that in this informal presentation, we are using the names *class*, *type*, *group*, etc, with their informal meanings, which refer to collections of things.

From this point on, let us name this new operation *sharing product*, and remark that the traditional product should appear as a particular case of the sharing product.

2 The Formal Presentation

2.1 Brief Description of the Framework

Before starting the formal presentation, let us make an overview of the categories involved.

The *sharing product* is defined considering a category that represents the syntactical class of the entities. Let us call $\mathcal{C}$ this category, and refer to its objects by the letters $A, B, C, D, \ldots$, using subscripts if necessary. Ordinary morphisms of $\mathcal{C}$ will be noted by s, possibly with subscripts. Special morphisms of $\mathcal{C}$ will be noted by the letters r and c, again with subscripts, if necessary. Finally, id_A, id_B, will be used for the identities. The syntactical category must have everything that is necessary to write a syntactical description. For the moment, as we are not yet concerned to attach semantics to its entities, let us say that $\mathcal{C}$ must have product and co-product for all objects, as well as the terminal object, so that we can talk about elements. As $\mathcal{C}$ has products for all objects, the family of objects $V^i, i < \omega$ belongs to the category. As it might be expected, the objects of $\mathcal{C}$ represent syntactical parts of a construction, that is, the types of terms. The n-uples V^i will be used to remark that a syntactical construct can be built using i components V.

There are two situations that must be expressed over the features in $\mathcal{C}$. The next two paragraphs sketch both.

If it is possible to write a term using i components, the same term can also be written using j components, where $j > i$, for we can always ignore the $j - i$ components. The reverse of this also makes sense: if a term is written using j components, where $j - i$ components are not used, then it can also be written with i components. To cope with these reductions and expansions, we will consider special morphisms of $\mathcal{C}$ that will be noted r.

If a term is built with i components, and another one is built with j components, then the two of them together are built with $i+j$ components. But if one component of the first must be the same as one component of the second then the two of them together are built with $i+j-1$ components. Let us call this situation *sharing*, and use a pair (c_1, c_2) of $\mathcal{C}$ morphisms to express this fact.

Now, to describe entities we will use morphisms of $\mathcal{C}$ in the following way: if P is an object of $\mathcal{C}$, and P is built over i components V, then P is represented by a morphism from V^i to P. In some sense, it is as if we were detaching components V of the description of the object. So, a subset of $Hom_{\mathcal{C}}(V^i, P)$ gives a set of terms of type P that can be constructed over, at most, i components. We say *at most* because some components may not be used.

Considering all the objects of $\mathcal{C}$ that can be written using at most i components, we have a category whose objects are subsets of $Hom_{\mathcal{C}}(V^i, A)$, for all A, object of $\mathcal{C}$, and whose morphisms are the usual morphisms between $hom-sets$, when the first argument is fixed.

As we have several values of i, we have several of these categories. Using the $\mathcal{C}$ morphisms r, we can define natural ways of passing from one to another, expanding or reducing the number of unused components. With all this, we have a framework to represent all the terms.

Now, we turn to the formal presentation.

2.2 The Formal Presentation

Note : In the following, let us reserve the letters i, j, k, n, m to be variables for natural numbers.

2.2.1 The Syntactical Category

Definition 2.2.2 *Category $\mathcal{C}$*
A syntactical category $\mathcal{C}$ is a locally small category with product, co-product and terminal object.

Definition 2.2.3 *Reductor in $\mathcal{C}$*
Given $\mathcal{C}$, a syntactical category, and $V \in Obj(\mathcal{C})$, a reductor r in $\mathcal{C}$ is a morphism $r : V^j \to V^i$ defined over the algebra $< \{id_V, 1_V\, tw\}, \times, \circ\} >$, where

$id_V : V \to V$, *is the identity morphism,*
$1_V : V \to 1$,
$tw : V \times V \to V \times V$
$\quad (v_1, v_2) \longmapsto (v_2, v_1)$,
$\times$ *is the usual product within $\mathcal{C}$ and*
$\circ$ *is the usual composition in $\mathcal{C}$.*

Definition 2.2.4 *Complementary in $\mathcal{C}$*
A pair of $\mathcal{C} - Morphisms$ $(r_1 : V^{i+j} \to V^i, r_2 : V^{i+j} \to V^j)$ is said to be

complementary if each morphism is a reductor, and, for $1 \leq n \leq i+j$, $(\pi_n \circ r_1 = id_V$ *and* $\pi_n \circ r_2 = 1_V)$, *or* $(\pi_n \circ r_2 = id_V$ *and* $\pi_n \circ r_1 = 1_V)$.

Example: r_1 and r_2 below are complementary

$r_1 = id_V \times 1_V \times id_V \times 1_V \times id_V$

$r_2 = 1_V \times id_V \times 1_V \times id_V \times 1_V$

that is, one preserves the components of V^5 that the other does not.

Definition 2.2.5 *Sharing in* $\mathcal{C}$

Given $\mathcal{C}$, *a syntactical category, and* $V \in Obj(\mathcal{C})$, *a sharing between* V^i *and* V^j *is a pair of reductor morphisms* $(c_1 : V^{i+j-1} \rightarrow V^i, c_2 : V^{i+j-1} \rightarrow V^j)$.

Given a sharing (c_1, c_2) *and a pair* (m, n) *such that* $m \leq i$ *and* $n \leq j$, *we say that* (c_1, c_2) *shares* (m, n) *if* $\pi_m \circ c_1 = \pi_n \circ c_2$.

Example: The pair (c_1, c_2), where c_1 and c_2 are described below, share (2,2).

$c_1 : V^6 \longrightarrow V^4$

$c_1 = id_V \times id_V \times id_V \times id_V \times 1_V \times 1_V$

$< a, b, c, d, e, f > \longmapsto < a, b, c, d >$

$c_2 : V^6 \longrightarrow V^3$

$c_2 = (tw \times id_V) \circ (1_V \times id_V \times 1_V \times 1_V \times id_V \times id_V)$.

$< a, b, c, d, e, f > \longmapsto < b, e, f > \longmapsto < e, b, f >$

2.2.6 The Functors H_{V^i}

Theorem 2.2.1 *Functors* H_{V^i}

Let $V \in Obj(\mathcal{C})$. H_{V^i} *is a family of functors defined by:*

$$\begin{array}{lll} H_{V^i} : \mathcal{C} & \longrightarrow & \mathcal{S}et \\ A & \longmapsto & Hom_{\mathcal{C}}(V^i, A), \text{ where } A \text{ has zero operators} \\ A \times B & \longmapsto & H_{V^i}(A) \times H_{V^i}(B) \\ A + B & \longmapsto & H_{V^i}(A) + H_{V^i}(B) \\ s : A \rightarrow B & \longmapsto & H_{V^i}(A) \rightarrow H_{V^i}(B) \\ & & f \longmapsto s \circ f \end{array}$$

Proof: H_{V^i} *is a functor.*

(i) H_{V^i} *preserves identities:*

$$\begin{array}{lllllll} H_{V^i}(id_A) & = & H_{V^i}(A) \rightarrow H_{V^i}(A) & = & H_{V^i}(id_A) \rightarrow H_{V^i}(id_A) & = & id_{H_{V^i}(A)} \\ & & f \longmapsto id_A \circ f & = & f \longmapsto f & & \end{array}$$

(ii) H_{V^i} *preserves composition: Let* $s_1 : A \rightarrow B$ *and* $s_2 : B \rightarrow C$ *be morphisms of* $\mathcal{C}$.

$$\begin{array}{lllll} H_{V^i}(s_2 \circ s_1) & = & H_{V^i}(A) \rightarrow H_{V^i}(C) & = & H_{V^i}(A) \rightarrow H_{V^i}(B) \rightarrow H_{V^i}(C) \quad = \\ & & f \longmapsto s_2 \circ s_1 \circ f & = & f \longmapsto s_1 \circ f \longmapsto s_2 \circ s_1 \circ f \\ & & & & \\ & = & H_{V^i}(s_2) \circ H_{V^i}(s_1) & & \end{array}$$

•

Proposition 2.2.2 *$H_{V^i}(A) + H_{V^i}(B) \subseteq Hom_{\mathcal{C}}(V^i, A+B)$ and $H_{V^i}(A) \times H_{V^i}(B) \subseteq Hom_{\mathcal{C}}(V^i, A \times B)$.*

Proof: *Let op be an abbreviation for $+$ or $\times$. Let us prove by induction on the number of operators that*

$$H_{V^i}(A) \; op \; H_{V^i}(B) \subseteq Hom_{\mathcal{C}}(V^i, A \; op \; B)$$

For zero operators in A and B:

$$(i) H_{V^i}(A) + H_{V^i}(B) = \quad Hom_{\mathcal{C}}(V^i, A) + Hom_{\mathcal{C}}(V^i, B) \subseteq \quad Hom_{\mathcal{C}}(V^i, A+B)$$

The last step is due to 4.0.5.

$$(ii) H_{V^i}(A) \times H_{V^i}(B) = \quad Hom_{\mathcal{C}}(V^i, A) \times Hom_{\mathcal{C}}(V^i, B) = \quad Hom_{\mathcal{C}}(V^i, A \times B) \subseteq$$

$$\subseteq Hom_{\mathcal{C}}(V^i, A \times B)$$

The third step is due to 4.0.2

Let the number of operators in A, B and op equals n, and $H_{V^i}(A) \; op \; H_{V^i}(B) \subseteq Hom_{\mathcal{C}}(V^i, A \; op \; B)$.

Let the number of operators of C be zero, then $H_{V^i}(A \; op \; B) \; op \; H_{V^i}(C)$ has $n+1$ operators.

$$H_{V^i}(A \; op \; B) \; op \; H_{V^i}(C) = \quad H_{V^i}(A) \; op \; H_{V^i}(B) \; op \; H_{V^i}(C) \subseteq$$

$$\subseteq Hom_{\mathcal{C}}(V^i, A \; op \; B) \; op \; Hom_{\mathcal{C}}(V^i, C) \subseteq \quad Hom_{\mathcal{C}}(V^i, (A \; op \; B) \; op \; C)$$

The third step comes from $H_{V^i}(A) \; op \; H_{V^i}(B) \subseteq Hom_{\mathcal{C}}(V^i, A \; op \; B)$, and the last step comes from 4.0.5 or 4.0.2

•

Note : What makes a functor H_{V^i} different from $Hom_{\mathcal{C}}(V^i, \_\,)$ is precisely the co-product line, as $Hom_{\mathcal{C}}(V^i, A) + Hom_{\mathcal{C}}(V^i, B) \ngeqq Hom_{\mathcal{C}}(V^i, A+B)$ (see 4.0.5). The object $H_{V^i}(A+B)$ does not contain morphisms from V^i to $A+B$ that sends elements V^i for both A and B.

Proposition 2.2.3 *A reductor $r : V^j \to V^i$ in $\mathcal{C}$ defines a natural transformation σ between the functors H_{V^i} and H_{V^j}, where each component is defined by:*

$$\begin{array}{rccc} \sigma_A : & H_{V^i}(A) & \to & H_{V^j}(A) \\ & f & \longmapsto & f \circ r \end{array}$$

Proof: *Given $s : A \to B$, a morphism of $\mathcal{C}$, recalling of the functors H_{V^i} and H_{V^j}, we have,*

$$\begin{array}{rccc} H_{V^i}(s): & H_{V^i}(A) & \to & H_{V^i}(B) \\ & f & \longmapsto & s \circ f \\ H_{V^j}(s): & H_{V^j}(A) & \to & H_{V^j}(B) \\ & f & \longmapsto & s \circ f \end{array}$$

Thus,

$$\begin{array}{rcccl} \sigma_B \circ H_{V^i}(s) = & H_{V^i}(A) & \to & H_{V^j}(B) = & H_{V^j}(s) \circ \sigma_A \\ & f & \longmapsto & s \circ f \circ r & \end{array}$$

That is, the following diagram commutes.

$$\begin{tikzcd} A \arrow[d, "s"'] & H_{V^i}(A) \arrow[r, "\sigma_A"] \arrow[d, "H_{V^i}(s)"'] & H_{V^j}(A) \arrow[d, "H_{V^j}(s)"] \\ B & H_{V^i}(B) \arrow[r, "\sigma_B"] & H_{V^j}(B) \end{tikzcd}$$

•

Note : When passing from $H_{V^i}(A)$ to $H_{V^j}(A)$ by such natural transformations, $j - i$ variables are added to the domain of each entity, but they are *not used*, as the following diagram commutes. That is, the meaning of the entity remains the same.

$$\begin{tikzcd} V^i \arrow[dr, "f"'] & V^j \arrow[l, "r"'] \arrow[d, "f \circ r"] \\ & A \end{tikzcd}$$

Definition 2.2.7 *Given σ, a natural transformation, γ is the natural transformation whose components are the inverse of σ's components. We call γ the inverse of σ. For each object A of $\mathcal{C}$,*

$$\gamma_A : \quad \sigma_A(H_{V^i}(A)) \quad \rightarrow \quad H_{V^i}(A)$$

and

$$\gamma_A \circ \sigma_A = Id_{H_{V^i}(A)}.$$

2.2.8 Categories $H_{V^i}(\mathcal{C})$

Theorem 2.2.4 *Categories $H_{V^i}(\mathcal{C})$*
Each functor H_{V^i}, for $i < \omega$, defines a category $H_{V^i}(\mathcal{C})$, whose objects are the subsets of $H_{V^i}(A)$, for each $A \in Obj(\mathcal{C})$, and morphisms are as p below:

$$\begin{array}{cccc} p: & H_{V^i}(A) & \rightarrow & H_{V^i}(B) \\ & f & \longmapsto & s \circ f \end{array}$$

where $s : A \rightarrow B \in Morf(\mathcal{C})$

Proof: *$H_{V^i}(\mathcal{C})$ is a category.*

(i) $H_{V^i}(\mathcal{C})$ has associative composition.

Let $s_1 : A \rightarrow B$, $s_2 : B \rightarrow C$ and $s_1 : C \rightarrow D$, morphisms of $\mathcal{C}$, and,

$$\begin{array}{cccc} p: & a & \rightarrow & b \\ & f & \longmapsto & s_1 \circ f \\ q: & b & \rightarrow & c \\ & g & \longmapsto & s_2 \circ g \\ t: & c & \rightarrow & d \\ & h & \longmapsto & s_3 \circ h \end{array}$$

Where $a \subseteq H_{V^i}(A), b \subseteq H_{V^i}(B), c \subseteq H_{V^i}(C)$ and $d \subseteq H_{V^i}(D)$. Then,

$$t \circ (q \circ p) \quad = \quad \begin{array}{c} t \circ (a \rightarrow c) \\ f \longmapsto s_2 \circ s_1 \circ f \end{array} \quad = \quad \begin{array}{c} a \rightarrow d \\ f \longmapsto s_3 \circ s_2 \circ s_1 \circ f \end{array} \quad =$$

$$= \quad \begin{array}{c} (b \rightarrow d) \circ p \\ g \longmapsto s_3 \circ s_2 \circ g \end{array} \quad = \quad (t \circ q) \circ p$$

(ii) $H_{V^i}(\mathcal{C})$ has identity for each object:

$$\begin{array}{cc} Id_b : & b \rightarrow b \\ & f \longmapsto id_B \circ f \end{array}$$

where $id_B : B \rightarrow B \in Morph(\mathcal{C})$ and b satisfies the two properties below:
Property 1: For each a and $p : a \rightarrow b$,

$$Id_b \circ p \quad = \quad \begin{array}{c} a \rightarrow b \rightarrow b \\ f \longmapsto s_1 \circ f \longmapsto id_B \circ s_1 \circ f \end{array} \quad = \quad p$$

Property 2: Likewise, for each c and $q : b \to c$, $q \circ Id_b = q$
•

Theorem 2.2.5 *Given two functors H_{V^i} and $H_{V^{i+1}}$, and a natural transformation $\sigma_{(i,i+1)} : H_{V^i} \to H_{V^{i+1}}$, there is a functor $\sigma_{(i,i+1)}$ defined from the image of H_{V^i} to the image of $H_{V^{i+1}}$ in the following way:*
For objects:

$$\begin{array}{cccc} \sigma_{(i,i+1)} : & H_{V^i}(\mathcal{C}) & \longrightarrow & H_{V^{i+1}}(\mathcal{C}) \\ & H_{V^i}(A) & \longmapsto & cod({\sigma_{(i,i+1)}}_A) \\ & f & \longmapsto & f \circ r \end{array}$$

where $r : V^{i+1} \to V^i$ is the reductor of the natural transformation $\sigma_{(i,i+1)}$. For morphisms:

$$\begin{array}{cccc} \sigma_{(i,i+1)} : & H_{V^i}(\mathcal{C}) & \longrightarrow & H_{V^{i+1}}(\mathcal{C}) \\ & H_{V^i}(A) \to H_{V^i}(B) & \longmapsto & cod({\sigma_{(i,i+1)}}_A) \to cod({\sigma_{(i,i+1)}}_B) \\ & f \longmapsto s \circ f & \longmapsto & f \circ r \longmapsto s \circ f \circ r \end{array}$$

where $r : V^{i+1} \to V^i$ is the reductor of the natural transformation $\sigma_{(i,i+1)}$, and s is a syntactical morphism $s : A \to B$.

Proof: *The proof is a consequence of 4.0.1.*
•

Note (i)**:** To extend theorem 2.2.5 to the categories $H_{V^i}(\mathcal{C})$ and $H_{V^i+1}(\mathcal{C})$ (instead of the image of functors), we consider for each subset X of $H_{V^i}(A)$ the restriction of ${\sigma_{(i,i+1)}}_A$ to X.

Note (ii)**:** From this point on, the context will make clear weather σ is the functor or the natural transformation.

Proposition 2.2.6 *$\sigma_{(i,j)}$ preserves product and co-product*
Let $a \subseteq H_{V^i}(A)$ and $b \subseteq H_{V^i}(B)$. Then, $\sigma_{(i,j)}(a \times b) = \sigma_{(i,j)}(a) \times \sigma_{(i,j)}(b)$, and $\sigma_{(i,j)}(a + b) = \sigma_{(i,j)}(a) + \sigma_{(i,j)}(b)$.

Proof: *From the product properties, for each $f \in a$ and $g \in b$, we have $< f, g > \circ r =< f \circ r, g \circ r >$. Similar for the co-product.*
•

2.2.9 Isomorphism in $H_{V^i}(\mathcal{C})$

Definition 2.2.10 *Isomorphism in $H_{V^i}(\mathcal{C})$*
A morphism $p : a \to b$ is an isomorphism if exists a morphism $q : b \to a$ such that $p \circ q = Id_b$ and $q \circ p = Id_a$. We call q the inverse of p.

Definition 2.2.11 *Isomorphic objects*

An object a is isomorphic to another object b (we write $a \simeq b$) if there is an isomorphism $p : a \to b$.

Proposition 2.2.7 *Let $a \subseteq H_{V^i}(A)$ and $b \subseteq H_{V^i}(B)$. Then, $a \simeq b$ iif $A \simeq B$*

Proof: *Suppose that $a \simeq b$ and p, q are the isomorphism and its inverse. Then we have $p \circ q(f) = Id_b(f)$ for all $f \in b$. Let s_p and s_q be the syntactical morphisms that define p and q. Then we have $s_p \circ s_q \circ f = f$, and thus $s_p \circ s_q = id_B$. Similar for $q \circ p$. Similar for the opposite direction.*

•

2.2.12 The Co-limit $H_{V^\omega}(\mathcal{C})$

Definition 2.2.13 *Co-limit of $H_{V^i}(\mathcal{C})$*

Consider the category whose objects are $H_{V^i}(\mathcal{C})$, and the morphisms are induced by the natural transformation between the functors H_{V^i}. $H_{V^\omega}(\mathcal{C})$ is the co-limit of the diagram composed by the objects of this category and the morphisms $\sigma_{(i,i+1)} : H_{V^i}(\mathcal{C}) \to H_{V^{i+1}}(\mathcal{C})$.

Theorem 2.2.8 *All objects of $H_{V^\omega}(\mathcal{C})$ come from a $H_{V^i}(\mathcal{C})$*
Let $a \in H_{V^\omega}(\mathcal{C})$, then there exists a' and i such that $a' \in H_{V^i}(\mathcal{C})$ and $a = \sigma_{(i,\omega)}(a')$

Proof: *Suppose that $a \in H_{V^\omega}(\mathcal{C})$, and for all a', i we have $a \neq \sigma_{(i,\omega)}(a')$. let $\mathcal{H}$ be the following category:*

$$Obj(\mathcal{H}) = Obj(H_{V^\omega}(\mathcal{C})) - \{a\}$$
$$Morph(\mathcal{H}) = Morph(H_{V^\omega}(\mathcal{C})) - \{f : Dom(f) = a \text{ or } Cod(f) = a\}$$

Both $\mathcal{H}$ and $H_{V^\omega}(\mathcal{C})$ are co-cones, but there is a unique $\sigma : \mathcal{H} \to H_{V^\omega}(\mathcal{C})$, thus $\mathcal{H}$ would be the co-limit, instead of $H_{V^\omega}(\mathcal{C})$.

•

2.2.14 Examples

(i) To represent single variables, one can use singletons of element morphisms in $\mathcal{C}$. For example, given $V \in Obj(\mathcal{C})$, and $a : 1 \to V \in Morph(\mathcal{C})$, $\{a\}$ is the object of the category $H_{V^0}(\mathcal{C})$ that represents the variable a.

(ii) To represent the domain of all variables, identities can be used: $id_V : V \to V$ is the object of the category $H_{V^1}(\mathcal{C})$ that represents variables.

(iii) Expressions built over one variable can be expressed within the category $H_{V^1}(\mathcal{C})$, for example,

$$\left\{ \begin{array}{lcl} f_1 : V \longrightarrow E & , & f_2 : V \longrightarrow E \quad , \quad ... \\ x \longmapsto x+1 & & x \longmapsto x+2 \end{array} \right\}$$

is a domain of expression with one variable.

(iv) The same domain can be viewed in the category $H_{V^2}(\mathcal{C})$, as the above functions can be expressed with two variables:

$$\left\{ \begin{array}{lcl} f'_1 : V \times V \longrightarrow E & , & f'_2 : V \times V \longrightarrow E \quad , \quad ... \\ (x,y) \longmapsto x+1 & & (x,y) \longmapsto x+2 \end{array} \right\}$$

(v) The functor $\sigma_{(1,2)}$ make possible this transference.

$$\begin{array}{rc} \sigma_{(1,2)} : & H_{V^1}(\mathcal{C}) \to H_{V^2}(\mathcal{C}) \\ & H_{V^1}(E) \to H_{V^2}(E) \\ & f \longmapsto f \circ r \end{array}$$

where r is the reductor $id_V \times 1_V$.

(vi) The functor $\sigma_{(i,\omega)}$, that is, the ideal compositions $... \circ \sigma_{(n,n+1)} \circ ... \circ \sigma_{(2,3)} \circ \sigma_{(1,2)}$ make possible to express any domain with functions 'built over ω variables'.

(vii) Given the domain of variables and the domain of expressions, both constructed over the same number of variables, say 1, we can form the domain of commands built over one variable using the syntactical morphism $s : V \times E \to C$:

$$\begin{array}{c} H_{V^1}(V) \times H_{V^1}(E) = H_{V^1}(V \times E) \\ H_{V^1}(V \times E) \longrightarrow H_{V^1}(C) \\ f \longmapsto s \circ f \end{array}$$

(viii) From the previous items, we can have a stratified view of the syntactical category. The more abstract view is $H_{V^\omega}(\mathcal{C})$, where the number of used variables is not important. The more concrete view is the smallest i that we can have in $H_{V^i}(\mathcal{C})$.

Note : Using programming language as motivation, let $P \in Obj(\mathcal{C})$ represent programs and $V \in Obj(\mathcal{C})$ represent variables. $H_{V^\omega}(P)$ is the object of $H_{V^\omega}(\mathcal{C})$ that represents the possibility of a program to use as many variables as necessary, since it uses a finite number. So, $H_{V^\omega}(\mathcal{C})$ is the more accurate category to represent programming language terms.

2.3 Brief Description of the Operations

Considering the framework presented in the previous sections, let us define the operations to be performed on it. As one might expect, all the operations that can be performed in the syntactical category must also have an analogous here. Thus, we have the product, which will be also referred to as *conventional product*, the co-product, and the new ones, the sharing and non-sharing product. As we have already commented, the conventional product turns out to be a special case of the sharing product. We will make this point clear in this section.

Recall that, in the syntactical category, the operations are performed over groups of entities (that is, syntactical classes), and not just over simple entities. As an example, blocks in a programming language can be expressed as the class of entities formed by the pairs *(command,command)*, that is, members of the product of syntactical class of *commands* and the syntactical class of *commands*. The same situation is pictured in this new framework, since an object $H_{V^i}(A)$ in a fixed category $H_{V^i}(\mathcal{C})$ denotes a subset of syntactical class of A: entities of A that can be built with at most i variables.

Starting by the non-sharing product, and still using programming language as motivation, let us consider, for example, C and V, as the objects of $\mathcal{C}$ that represent the syntactical class of commands and variables. Let us also say that C is $C \times C$, or C_0 is $C_1 \times C_2$, to make references clear.

The point is, what is the non-sharing product of $H_{V^i}(C_1)$ and $H_{V^i}(C_2)$ supposed to result? Although the functor H_{V^i} says that $H_{V^i}(C_1 \times C_2) = H_{V^i}(C_1) \times H_{V^i}(C_2)$, there is nothing indicating that both C's use the same variables. If one have a command constructed over i variables (ex: $f : x \rightarrow x := x + 1$) and another command constructed over i variables (ex: $g : x \rightarrow x := x + 2$), then, if both are put together (side by side), one will not have a command constructed over i variables any more. The result should be a command constructed over $i + i$ variables, as the variables used by the first can be different from those used by the second (not $< f, g >: x \rightarrow < x := x + 1, x := x + 2 >$, but $< f, g >: < x, y > \rightarrow < x := x + 1, y := y + 2 >$). So, the non-sharing product is not exactly the usual product in subsets of $Hom_{\mathcal{C}}(V^i, \_)$ (see 4.0.2 and 4.0.3 to make clear the product in hom-sets).

At first sight, this seems to be a problem: the 'wanted' product of two objects of H_{V^i} does not belong to H_{V^i}. However, we can always transfer the objects we are dealing with to the category $H_{V^\omega}(\mathcal{C})$, as it has all, and no more than the objects of $H_{V^i}(\mathcal{C})$, for all i.

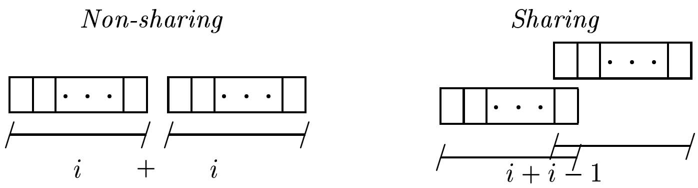
Thus, we will perform the non-sharing product in the following way:

> *To have the non-sharing product of two objects constructed over i components, look at each one of them as if they were constructed over $i + i$ components and, then, use the conventional product to operate them.*

Note that this step of expanding the number of components of an object from i to $i+i$ must be done in a way that does not change the size of the object. That is, if the object is a class composed by two entities of i components, it has to become a class composed by two entities of $i+i$ components.

This outlines the non-sharing product between objects with the same number of components. The general case, different numbers of components, is very similar, so will be presented only in the Formal Presentation Section.

Bearing in mind that the non-sharing product works as putting entities side by side, let us turn to the sharing product. The idea is the same, but components of each entity that are to be shared must be overplaced, as shows the following picture:



Thus,

> *To have the sharing product of two objects constructed over i components, look at each one of them as if they were constructed over $i+i-1$ components and, then, use the conventional product to operate them.*

Note that we are making possible the sharing of *one* component of one entity with *one* component of the other.

Finally, the co-product. If the two arguments of the co-product are constructed over the same number of components, there is not much to be said about the co-product, for the conventional co-product produces the expected result. To make it clear, $H_{V^i}(A) + H_{V^i}(B) = H_{V^i}(A+B)$ is stating that if A is constructed with i components, and B is constructed with i components then, if one has i components, one can construct A or B. In the general case, however, it will be necessary to make some adjustments in order to operate objects with different numbers of components.

2.4 The Formal Presentation

2.4.1 Non-Sharing Product

Definition 2.4.2 *Non-sharing product from a fixed category $H_{V^i}(\mathcal{C})$*
Given $a, b \in Obj(H_{V^i}(\mathcal{C}))$, the non-sharing product $a \boxed{\mathrm{ns}}_{(i+i)} b$ is defined as the following object of $H_{V^{i+i}}(\mathcal{C})$:

$$a \boxed{\mathrm{ns}}_{(i+i)} b = \sigma_{(i,i+i)}(a) \times \sigma'_{(i,i+i)}(b),$$

where $\sigma_{(i,i+i)}$ *and* $\sigma'_{(i,i+i)}$ *are defined over complementary reductors* r *and* r'*.*
The first projection is

$$\begin{array}{llcccc}
\pi_1 : & (a \boxed{\scriptstyle ns}_{(i+i)} b) & \stackrel{p}{\longrightarrow} & \sigma_{i,i+i}(a) & \stackrel{\gamma_{(i,i+i)}}{\longrightarrow} & a \\
& < f \circ r, g \circ r' > & \longmapsto & \pi'_1 \circ < f \circ r, g \circ r' >= f \circ r & \longmapsto & f
\end{array}$$

where π'_1 *is the projection in the syntactical category* $\mathcal{C}$*, and* $\gamma_{(i+i,i)}$ *is the inverse of* $\sigma_{(i,i+i)}$*.*
The second projection is straightforward.

Note (i)**:** As usual, $\times$ means the conventional product.

Note (ii)**:** The requirement of using complementary reductors in $\mathcal{C}$ ensures that the components used in one entity are not used in the other.

Note (iii)**:** One can always omit the subscript of the operator as we can deduce the category where the product will be performed from the category to which the arguments belong. As a remark, however, we are going to preserve the subscripts in this section.

2.4.3 Sharing Product

Definition 2.4.4 *Sharing product from a fixed category* $H_{V^i}(\mathcal{C})$
Given $a, b \in Obj(H_{V^i}(\mathcal{C}))$*, the sharing product* $a \boxed{\scriptstyle s}_{(i+i-1)} b$ *is defined as the following object of* $H_{V^{i+i-1}}(\mathcal{C})$*:*

$$a \boxed{\scriptstyle s}_{(i+i-1)} b = \sigma_{(i,i+i-1)}(a) \times \sigma'_{(i,i+i-1)}(b),$$

where $\sigma_{(i,i+i-1)}$ *and* $\sigma'_{(i,i+i-1)}$ *are defined over a sharing pair* (c, c')*.*
The first projection is

$$\begin{array}{llcccc}
\pi_1 : & (a \boxed{\scriptstyle s}_{(i+i-1)} b) & \stackrel{p}{\longrightarrow} & \sigma_{i,i+i-1}(a) & \stackrel{\gamma_{(i,i+i-1)}}{\longrightarrow} & a \\
& < f \circ c, g \circ c' > & \longmapsto & \pi'_1 \circ < f \circ c, g \circ c' >= f \circ c & \longmapsto & f
\end{array}$$

where π'_1 *is the projection in the syntactical category* $\mathcal{C}$*, and* $\gamma_{(i+i-1,i)}$ *is the inverse of* $\sigma_{(i,i+i-1)}$*.*
The second projection is straightforward.

Note (i)**:** As usual, $\times$ means the conventional product.

Note (ii)**:** Sharing pairs ensures that one component is used by both entities. The other components are either used in one entity or in the other.

Note (iii)**:** Again, the subscript of the operator can be omitted.

2.4.5 Co-Product

Definition 2.4.6 *Co-product in a fixed category $H_{V^i}(\mathcal{C})$*
Given $a \subseteq H_{V^i}(A)$ and $b \subseteq H_{V^i}(B)$, the co-product $a \boxed{\text{co}}\, b$ is defined as the object $a + b$ with the usual injections.

2.4.7 Operations in the Co-limit Category

It can be noted from the previous definitions that the sharing and non-sharing product of objects of a category $H_{V^i}(\mathcal{C})$ yield an object that considers a number of components greater than i, that is, an object of a category $H_{V^j}(\mathcal{C})$ with $j > i$. In order to have all the objects in the same category, we can get their corresponding in the co-limit category $H_{V^\omega}(\mathcal{C})$. So, there are two possibilities. The first is to transfer the objects to the category $H_{V^\omega}(\mathcal{C})$ and, then, to perform the operation. The second is to perform the operation and, then, to transfer the objects to the category $H_{V^\omega}(\mathcal{C})$. Both approaches are valid, since the functors σ preserve product and co-product (as proved in 2.2.6). We choose, however, the second approach, as it is easier to handle objects with small numbers of components, both in setting the definitions and in presenting examples.

Let us organise this presentation in the following way. First we generalise the operators' definition in order to get, as operands, objects of different categories. To be more precise, operands belong to the same category as the operator, but they may have been transferred from another category via the functors σ. Then, we re-define the operators' definitions considering only the co-limit category. Finally, we prove that the operations in co-limit are well defined, that is, one may choose an object of the category $H_{V^i}(\mathcal{C})$ to operate, or its corresponding in an $H_{V^j}(\mathcal{C})$, for any j. The result in $H_{V^\omega}(\mathcal{C})$ will be the same for i or j. Such proof, however, will be presented only for the non-sharing product, as, for the other operators it is straightforward.

Definition 2.4.8 *Non-sharing product from different categories*
Given $a \in Obj(H_{V^i}(\mathcal{C}))$ and $b \in Obj(H_{V^j}(\mathcal{C}))$, the non-sharing product $a \boxed{\text{ns}}_{(i+j)} b$ is defined as the following object of $H_{V^{i+j}}(\mathcal{C})$:

$$a \boxed{\text{ns}}_{(i+j)} b = \sigma_{(i,i+j)}(a) \times \sigma_{(j,i+j)}(b),$$

where $\sigma_{(i,i+j)}$ and $\sigma_{(j,i+j)}$ are defined over complementary reductors r and r'. The first projection is

$$\begin{array}{lccccc} \pi_1 : & (a \boxed{\text{ns}}_{(i+j)} b) & \xrightarrow{p} & \sigma_{i,i+j}(a) & \xrightarrow{\gamma_{(i,i+j)}} & a \\ & < f \circ r, g \circ r' > & \longmapsto & \pi_1' \circ < f \circ r, g \circ r' > = f \circ r & \longmapsto & f \end{array}$$

where π_1' is the projection in the syntactical category $\mathcal{C}$, and $\gamma_{(i+j,i)}$ is the inverse of $\sigma_{(i,i+j)}$.
The second projection is straightforward.

Definition 2.4.9 *Non-sharing product in $H_{V^\omega}(\mathcal{C})$*
Given $a, b \in Obj(H_{V^\omega}(\mathcal{C}))$, the non-sharing product $a \boxed{\text{ns}} b$ is defined as the following object:

$$a \boxed{\text{ns}} b = \sigma_{(i+j,\omega)}(a' \boxed{\text{ns}}_{(i+j)} b'),$$

where $a' \in Obj(H_{V^i}(\mathcal{C}))$, $b' \in Obj(H_{V^j}(\mathcal{C}))$ and $\sigma_{(i+j,\omega)}$ is defined over a reductor r'' in $\mathcal{C}$.
The first projection is

$$\begin{array}{llclcl} \pi_1 : & (a \boxed{\text{ns}} b) & \xrightarrow{\gamma_{(i+j,\omega)}} & a' \boxed{\text{ns}} b' & \xrightarrow{\pi_{1_{(i,i+j)}}} & a \\ & < f \circ r, g \circ r' > \circ r'' & \longmapsto & < f \circ r, g \circ r' > & \longmapsto & f \end{array}$$

where $\pi'_{1_{(i+j)}}$ is the projection defined in 2.4.8, and $\gamma_{(i+j,\omega)}$ is the inverse of $\sigma_{(i+j,\omega)}$.
The second projection is straightforward.

Definition 2.4.10 *Sharing Product from different categories*
Given $a \in Obj(H_{V^i}(\mathcal{C}))$ and $b \in Obj(H_{V^j}(\mathcal{C}))$, the sharing product $a \boxed{\text{s}}_{(i+j-1)} b$ is defined as the following object of $H_{V^{i+j-1}}(\mathcal{C})$:

$$a \boxed{\text{s}}_{(i+j-1)} b = \sigma_{(i,i+j-1)}(a) \times \sigma_{(j,i+j-1)}(b),$$

where $\sigma_{(i,i+j-1)}$ and $\sigma_{(j,i+j-1)}$ are defined over complementary reductors r and r'.
The first projection is

$$\begin{array}{llclcl} \pi_1 : & (a \boxed{\text{ns}}_{(i+j-1)} b) & \xrightarrow{p} & \sigma_{i,i+j-1}(a) & \xrightarrow{\gamma_{(i,i+j-1)}} & a \\ & < f \circ r, g \circ r' > & \longmapsto & \pi'_1 \circ < f \circ r, g \circ r' > = f \circ r & \longmapsto & f \end{array}$$

where π'_1 is the projection in the syntactical category $\mathcal{C}$, and $\gamma_{(i+j-1,i)}$ is the inverse of $\sigma_{(i,i+j-1)}$.

The second projection is straightforward.

Definition 2.4.11 *Sharing Product in $H_{V^\omega}(\mathcal{C})$*
Given $a, b \in Obj(H_{V^\omega}(\mathcal{C}))$, the sharing product $a \boxed{\text{s}} b$ is defined as the following object:

$$a \boxed{\text{s}} b = \sigma_{(i+j-1,\omega)}(a' \boxed{\text{s}}_{(i+j-1)} b'),$$

where $a' \in Obj(H_{V^i}(\mathcal{C}))$, $b' \in Obj(H_{V^j}(\mathcal{C}))$ and $\sigma_{(i+j-1,\omega)}$ is defined over a reductor r'' in $\mathcal{C}$.
The first projection is

$$
\begin{array}{llcccc}
\pi_1 : & (a \boxed{\scriptstyle ns} b) & \xrightarrow{\gamma_{(i+j-1,\omega)}} & a' \boxed{\scriptstyle ns} b' & \xrightarrow{\pi_{1_{(i,i+j-1)}}} & a \\
 & < f \circ r, g \circ r' > \circ r'' & \longmapsto & < f \circ r, g \circ r' > & \longmapsto & f
\end{array}
$$

where $\pi'_{1_{(i+j-1)}}$ is the projection defined in 2.4.10, and $\gamma_{(i+j-1,\omega)}$ is the inverse of $\sigma_{(i+j-1,\omega)}$.

The second projection is straightforward.

Definition 2.4.12 *Co-product from different categories*
Given $a \subseteq H_{V^i}(A)$ and $b \subseteq H_{V^j}(B)$, with $j \geq i$, the co-product $a \boxed{\scriptstyle co}_j b$ is defined as the following object of $H_{V^j}(\mathcal{C})$:

$$a \boxed{\scriptstyle co}_j b = \sigma_{(i,j)}(a) + b,$$

where $\sigma_{(i,j)}$ is defined over a reductor r in $\mathcal{C}$.
The first injection is

$$
\begin{array}{rl}
i_1 : & a \to \sigma_{(i,j)}(a) \to a \boxed{\scriptstyle co}_j b \\
 & f \longmapsto f \circ r \longmapsto i'_1 \circ f \circ r
\end{array}
$$

where $i'_1 : A \to A + B$ is the injection in $\mathcal{C}$.
The second injection is

$$
\begin{array}{rl}
i_2 : & b \to a \boxed{\scriptstyle co}_j b \\
 & f \longmapsto i'_2 \circ f
\end{array}
$$

where $i'_2 : B \to A + B$ is the injection in $\mathcal{C}$.

Note (i): The requirement of using reductor morphism in $\mathcal{C}$ is only a trick to make both object belong to the same category.

Note (ii): Again, we can omit the subscript of the operator.

Definition 2.4.13 *Co-product in $H_{V^\omega}(\mathcal{C})$*
Given $a \subseteq H_{V^\omega}(A), b \subseteq H_{V^\omega}(B)$, the co-product $a \boxed{\scriptstyle co} b$ is defined as the following object:

$$a \boxed{\scriptstyle co} b = \sigma_{(j,\omega)}(a' \boxed{\scriptstyle co}_j b'),$$

where $a' \in Obj(H_{V^i}(\mathcal{C}))$ and $b' \in Obj(H_{V^j}(\mathcal{C}))$, $j \geq i$ and $\sigma_{(j,\omega)}$ is defined over a reductor r' in $\mathcal{C}$.

The first injection is

$$
\begin{array}{rcccccc}
 & & i_{1_j} & & \sigma_{j,\omega} & \\
i_1: & a & \to & a \boxed{\text{co}}_j b & \to & a \boxed{\text{co}}\, b \\
 & f & \longmapsto & i'_1 \circ f \circ r & \longmapsto & i'_1 \circ f \circ r \circ r'
\end{array}
$$

where i_{1_j} is the first injection defined in 2.4.12.
The second injection is

$$
\begin{array}{rcccccc}
 & & i_{2_j} & & \sigma_{j,\omega} & \\
i_2: & a & \to & a \boxed{\text{co}}_j b & \to & a \boxed{\text{co}}\, b \\
 & f & \longmapsto & i'_2 \circ f \circ r & \longmapsto & i'_2 \circ f \circ r \circ r'
\end{array}
$$

where i_{2_j} is the second injection defined in 2.4.12.

Theorem 2.4.1 *Non-sharing product is well defined. Let $k > j$, $a' \in Obj(H_{V^i}(\mathcal{C}))$, $b' \in Obj(H_{V^j}(\mathcal{C}))$ and $c' = \sigma_{(j,k)}(b')$. Then,*

$$a \boxed{\text{ns}}\, b = \sigma_{(i+j,\omega)}(a' \boxed{\text{ns}}\, b') = \sigma_{(i+k,\omega)}(a' \boxed{\text{ns}}\, c')$$

Proof: *To facilitate reading, we are underlining the part of the expression that is changed from one step to the other.*

$$
\begin{array}{ll}
a \boxed{\text{ns}}\, b = & \\
= \sigma_{(i+j,\omega)}(a' \boxed{\text{ns}}\, b') & \text{by the definition of } \boxed{\text{ns}} \text{ in } H_{V^\omega}(\mathcal{C}) \\
= \underline{\sigma_{(i+j,\omega)}}(\sigma_{(i,i+j)}(a') \times \sigma_{(j,i+j)}(b')) & \text{by the definition of } \boxed{\text{ns}} \text{ in } H_{V^{i+j}}(\mathcal{C}) \\
= \sigma_{(i+k,\omega)} \circ \underline{\sigma_{(i+j,i+k)}(\sigma_{(i,i+j)}(a') \times \sigma_{(j,i+j)}(b'))} & \text{by the composition and } j < k \\
= \sigma_{(i+k,\omega)} \circ (\underline{\sigma_{(i+j,i+k)} \circ \sigma_{(i,i+k)}(a')} \times \underline{\sigma_{(i+j,i+k)} \circ \sigma_{(j,i+j)}(b')}) & \text{by 2.2.6} \\
= \sigma_{(i+k,\omega)} \circ (\sigma_{(i,i+k)}(a') \times \underline{\sigma_{(j,i+k)}(b')}) & \text{by composition} \\
= \sigma_{(i+k,\omega)} \circ (\sigma_{(i,i+k)}(a') \times \sigma_{(k,i+k)} \circ \underline{\sigma_{(j,k)}(b')}) & \text{by composition and } j < k \\
= \sigma_{(i+k,\omega)} \circ (\sigma_{(i,i+k)}(a') \times \underline{\sigma_{(k,i+k)}(c')}) & \text{by the definition of } c' \\
= \sigma_{(i+k,\omega)}(a' \boxed{\text{ns}}\, c') & \text{by the definition of } \boxed{\text{ns}} \text{ in } H_{V^{i+k}}(\mathcal{C})
\end{array}
$$

•

Theorem 2.4.2 *Sharing product is well defined. Let $k > j$, $a' \in Obj(H_{V^i}(\mathcal{C}))$, $b' \in Obj(H_{V^j}(\mathcal{C}))$ and $c' = \sigma_{(j,k)}(b')$. Then,*

$$a \boxed{\text{s}}\, b = \sigma_{(i+j,\omega)}(a' \boxed{\text{s}}\, b') = \sigma_{(i+k,\omega)}(a' \boxed{\text{s}}\, c')$$

Theorem 2.4.3 *Co-product is well defined. Let $k > j$, $a' \in Obj(H_{V^i}(\mathcal{C}))$, $b' \in Obj(H_{V^j}(\mathcal{C}))$ and $c' = \sigma_{(j,k)}(b')$. Then,*

$$a \boxed{\text{co}}\, b = \sigma_{(i+j,\omega)}(a' \boxed{\text{co}}\, b') = \sigma_{(i+k,\omega)}(a' \boxed{\text{co}}\, c')$$

The prooves of 2.4.2 and 2.4.3 are similar to 2.4.1.

2.5 Example

Given a grammar, let us consider as Syntactical Category the category whose objects are the non-terminals of the grammar, the terminal object 1 and the ones resulting of products and co-products. Morphisms are the projections, injections etc, and the syntactical morphisms shown in the table below:

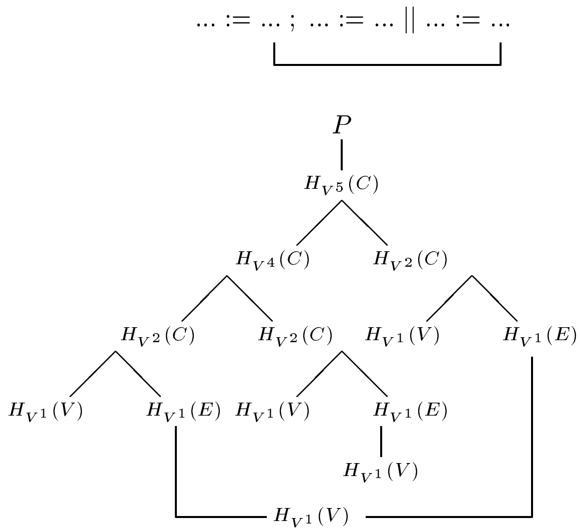
	BNF	*rules*		*Abstract*	*Syntax*	*Syntactical Morphisms*
1	$P ::=$	C		$P =$	C	$s_1 : C \to P$
2	$C ::=$	$C; C$	$\vert$	$C =$	$(C \times C)+$	$s_2 : C \times C \to C$
3		$C \vert\vert C$	$\vert$		$(C \times C)+$	$s_3 : C \times C \to C$
4		$V := E$			$(V \times E)$	$s_4 : V \times E \to C$
5	$E ::=$	$E * E$	$\vert$	$E =$	$(E \times E)+$	$s_5 : E \times E \to E$
6		V			V	$s_6 : V \to E$

By the third rule we want to express parallelism, but also state that sharing of variables is possible. Usually, there would be one more rule for commands, where the variables to be shared would be specified, but we are omitting such rule here, since we do not want to treat context sensitive information. What we have in the third rule is not the simple product, but the sharing one. In the table below, we re-write the productions using the notation and operators to stress the sharing in the syntax.

			Syntax expliciting sharing	
1	P	$::=$	$\sigma_{i,\omega}(H_{V^i}(C))$	
2	$H_{V^{i+j}}(C)$	$::=$	$H_{V^i}(C) \boxed{\text{ns}}\, H_{V^j}(C)$	$\boxed{\text{co}}$
3			$H_{V^i}(C) \boxed{\text{s}}\, H_{V^j}(C), (m, n)$	$\boxed{\text{co}}$
4			$H_{V^i}(V) \boxed{\text{ns}}\, H_{V^j}(E)$	
5	$H_{V^{i+j}}(E)$	$::=$	$H_{V^i}(E) \boxed{\text{ns}}\, H_{V^j}(E)$	$\boxed{\text{co}}$
6			$H_{V^i}(E)$	

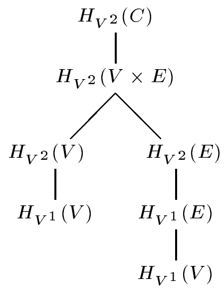
As usual the scope of names is restricted to the rule. This is also valid for i, j, m, and n. The first rule means: *a program is a command of i variables, seen as a command over an infinite number of variables.* In line 2, as well as in line 4, we have an object of the category $H_{V^{i+j}}(\mathcal{C})$ as result of these non-sharing products. In line 3, the sharing product results an object of the category $H_{V^{i+j-1}}(\mathcal{C})$. As the rule makes the co-product of them, we have the final object in the category $H_{V^{i+j}}(\mathcal{C})$. Thus, the second rule means: *a command built over $i + j$ variables is either the non-sharing product of a command of i variables with a command of j variables, or the sharing product of a command of i variables with a command of j variables, where the variables at positions m and n, respectively, are being shared, or, finally, the non-sharing product of a variable built over i variables with an expression of j variables.* The third rule is straightforward.

The next figure shows an instance of the syntactic graph of this grammar. The program pictured has two commands in parallel, the first one composed by two attributions. The program shares the second variable of the first command with the second variable of the second command.



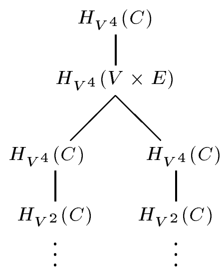
Let us give more details about the construction of this syntactic graph. In order to make the example simpler, we are going to consider the domain of variables containing only a single function: the identity. As we start with domains of size one, and make only products with them, we are going to end with a domain of programs of size one. To be faithful to the formal presentation we should get the whole $Hom_{\mathcal{C}}(V,V)$, but, simplifying in this way, we can have an idea of the sequence of composition until the construction of the program.

We start with two subsets of $H_{V^1}(V)$: $\{id_V\}$, and $\{id_V\}$. The second set must become a subset of $H_{V^1}(E)$, in order to form the attribution. The morphism $H_{V^1}(V) \to H_{V^1}(E)$ sends a function from V to V in the domain to a function from V to E, using the syntactical morphism $s_6 : V \to E$. So, we have $\{s_6 \circ id_V\}$. Now, performing the non-sharing product of both sets, we first transfer both for $H_{V^2}(\mathcal{C})$ and then use the product in sets. Thus we have $\{id_V \circ r_2\} \times \{s_6 \circ id_V \circ r_1\}$, where $r_2 = id_V \times 1_V$ and $r_1 = 1_V \times id_V$. This equals $\{< id_V \circ r_2, s_6 \circ id_V \circ r_1 >\}$. The morphism $H_{V^2}(V \times E) \to H_{V^2}(C)$ sends a function from V^2 to $V \times E$ in the domain to a function from V^2 to C, using the syntactical morphism $s_4 : V \times E \to C$, what yields $\{s_4 \circ < id_V \circ r_2, s_6 \circ id_V \circ r_1 >\}$. At this point we have already built the following part of the syntax graph:



that is summarised in the previous graph as the binary branch located in the left.

Assume that another branch like this has been constructed, giving rise to the following subset of $H_{V^2}(C)$: $\{s_4\circ < id_V \circ r_3, s_6 \circ id_V \circ r_4 >\}$, that is, another attribution. Again, performing the non-sharing product of this and the previous one, we first transfer both to $H_{V^4}(\mathcal{C})$ and then use the product in sets. Thus we have $\{s_4\circ < id_V \circ r_2, s_6 \circ id_V \circ r_1 > \circ r_5\} \times \{s_4\circ < id_V \circ r_3, s_6 \circ id_V \circ r_4 > \circ r_6\}$, where r_5 and r_6 are complementary . This equals $\{< s_4\circ < id_V \circ r_2, s_6 \circ id_V \circ r_1 > \circ r_5, s_4\circ < id_V \circ r_3, s_6 \circ id_V \circ r_4 > \circ r_6 >\}$, a subset of $H_{V^4}(C \times C)$. The morphism $H_{V^4}(C \times C) \to H_{V^4}(C)$ uses the syntactical morphism $s_2 : C \times C \to C$ to send a function from V^4 to $C \times C$ in the domain to a function from V^4 to C, what makes $\{s_2\circ < s_4\circ < id_V \circ r_2, s_6 \circ id_V \circ r_1 > \circ r_5, s_4\circ < id_V \circ r_3, s_6 \circ id_V \circ r_4 > \circ r_6 >\}$. Turning back to the syntax graph we have just built:



Now, let assume that we have another branch of attribution that is, a subset of $H_{V^2}(C)$: $\{s_4\circ < id_V \circ r_7, s_6 \circ id_V \circ r_8 >\}$, and we are going to make the sharing product of both, sharing the second variable of the first command with the second variable of the second program. We first transfer both to $H_{V^5}(\mathcal{C})$ and then use the product in sets. When passing from $H_{V^4}(\mathcal{C})$ and $H_{V^2}(\mathcal{C})$ to $H_{V^5}(\mathcal{C})$ it must be used a pair of sharing morphisms. Let us use the following pair:

$$c_1 : id_V \times id_V \times id_V \times id_V \times 1_V$$
$$c_1 : V^5 \longrightarrow V^4$$

$$< a, b, c, d, e > \longmapsto < a, b, c, d >$$

$$c_2 : tw \circ (1_V \times id_V \times 1_V \times 1_V \times id_V)$$
$$c_2 : V^5 \longrightarrow V^4$$
$$< a, b, c, d, e > \longmapsto < b, e > \longmapsto < e, b >$$

So, the sets are, now, $\{s_2 \circ < s_4 \circ < id_V \circ r_2, s_6 \circ id_V \circ r_1 > \circ r_5, s_4 \circ < id_V \circ r_3, s_6 \circ id_V \circ r_4 > \circ r_6 > \circ c_1\}$ and $\{s_4 \circ < id_V \circ r_7, s_6 \circ id_V \circ r_8 > \circ c_2\}$, that, after the product in sets, became $\{< s_2 \circ < s_4 \circ < id_V \circ r_2, s_6 \circ id_V \circ r_1 > \circ r_5, s_4 \circ < id_V \circ r_3, s_6 \circ id_V \circ r_4 > \circ r_6 > \circ c_1, s_4 \circ < id_V \circ r_7, s_6 \circ id_V \circ r_8 > \circ c_2 >\}$, an object of $H_{V^5}(C \times C)$. Note that the definition of $< c_1, c_2 >$ makes the second variable of the first command equal to the second variable of the second command. One more time, $C \times C$ is sent to c, by s_2, then we have $\{s_2 \circ < s_2 \circ < s_4 \circ < id_V \circ r_2, s_6 \circ id_V \circ r_1 > \circ r_5, s_4 \circ < id_V \circ r_3, s_6 \circ id_V \circ r_4 > \circ r_6 > \circ c_1, s_4 \circ < id_V \circ r_7, s_6 \circ id_V \circ r_8 > \circ c_2 >\}$, subset of $H_{V^5}(C)$. At last, the functor $\sigma_{(i,\omega)}$ expand the set of variables of this subset of $H_{V^5}(C)$ turning it into a program.

2.6 Last Remarks

We conclude this presentation with two additional remarks. First, stating relations among the three kinds of products that we have mentioned in this section: the conventional product (that is, product as sets), the non-sharing and the sharing product. Second, commenting the closure under the defined operations.

2.6.1 The Conventional Product

Regarding $hom - sets$, we have:

$Hom_{\mathcal{C}}(a, b) \times Hom_{\mathcal{C}}(a, c) \simeq Hom_{\mathcal{C}}(a, b \times c)$ (4.0.2),
$Hom_{\mathcal{C}}(a, b) \times Hom_{\mathcal{C}}(c, b) \simeq Hom_{\mathcal{C}}(a + c, b)$ (4.0.3), and the simplest case
$Hom_{\mathcal{C}}(a, b) \times Hom_{\mathcal{C}}(a, b) \simeq Hom_{\mathcal{C}}(a + a, b) \simeq Hom_{\mathcal{C}}(a, b \times b)$ (4.0.4),

but we do not know a way of expressing the product of $hom - sets$ $Hom_{\mathcal{C}}(a, b) \times Hom_{\mathcal{C}}(c, d)$, for $a \neq c$ and $b \neq d$, as a $hom - set$. As we do want to express within $H_{V^i}(\mathcal{C}), i < \omega$ the product of sets of functions with different domains and co-domains, the conventional product itself is not adequate. It has to be increased with mechanisms of moving operands to the same category. To the conventional product plus this mechanism we give the name *Non-Sharing Product*.

2.6.2 Sharing and Non-Sharing Product

It is not possible to state the differences between sharing and non-sharing product within the categories $H_{V^i}(\mathcal{C})$, since each operation will result an object in a different category. However, it is always possible to analyse operands and result as objects of $H_{V^\omega}(\mathcal{C})$. In order to have 'easy-to-handle' objects, instead of $H_{V^\omega}(\mathcal{C})$, let us choose the least category where we can find all the objects we are interested in. So, given $a \subseteq H_{V^i}(\mathcal{C})$ and $b \subseteq H_{V^i}(\mathcal{C})$, let us compare $a \boxed{\scriptstyle ns}_{i+j} b$ and $\sigma_{i+j-1,i+j}(a \boxed{\scriptstyle s}_{i+j-1} b)$. First of all, let us assume that functions in a really use all the i components V, that is, no σ functor has been applied. The same assumption is made for b.

Although the number of functions in each object is the same (as operands are the same), these objects are not isomorphic as the functions within each of which are essentially different. In $a \boxed{\scriptstyle ns}_{i+j} b$ functions use all the $i+j$ components, whereas in $\sigma_{i+j-1,i+j}(a \boxed{\scriptstyle s}_{i+j-1} b)$ there is an unused component among the $i+j$ ones. In fact, there is no way of having in $H_{V^{i+j}}(\mathcal{C})$ a morphism linking $a \boxed{\scriptstyle ns}_{i+j} b$ and $\sigma_{i+j-1,i+j}(a \boxed{\scriptstyle s}_{i+j-1} b)$ as syntactical morphisms do not capture this feature. Hence $a \boxed{\scriptstyle ns}_{i+j} b$ and $\sigma_{i+j-1,i+j}(a \boxed{\scriptstyle s}_{i+j-1} b)$ are isomorphic as sets, but not as $H_{V^{i+j}}(\mathcal{C})$ objects.

2.6.3 Closure Under Sharing, Non-Sharing Product and Co-Product

Recall the definition of the functors H_{V^i}, for $i < \omega$. Stating that $H_{V^i}(C_1 \times C_2) = H_{V^i}(C_1) \times H_{V^i}(C_2)$ this definition considers compound terms in C and constructs sets of compound terms in $H_{V^i}(\mathcal{C})$. Although bringing the desired closure for each category $H_{V^i}(\mathcal{C})$ under product and co-product, the definition of H_{V^i} is not compatible with the intuitive idea that, when sharing is not intended, putting together two terms with i components yields a term with $i+i$ components. For this reason, we ignore the conventional product and define the non-sharing producteven for operands belonging to the same category. When sharing is intended, the conventional product is still not adequate, as it enforces the sharing of *all* components (in $H_{V^i}(C_1 \times C_2)$ both C_1 and C_2 are written with the same i components). So, we also need the sharing product.

Being faithful to the intuitive idea, the new products have to yield a term of a different category, what implies in the non-closure of each $H_{V^i}(\mathcal{C})$ under non-sharing and sharing products. Similar situation also happens under co-product. For this reason, when we refer to the closure under non-sharing, sharing and co-products, the underline framework is not each $H_{V^i}(\mathcal{C})$, but all of them together. The closure of $H_{V^\omega}(\mathcal{C})$ under non-sharing, sharing and co-products is enought for making possible the construction of compound terms.

As $H_{V^\omega}(\mathcal{C})$ is the co-limit category and the morphisms $\sigma_{(i,\omega)}$, for $i < \omega$ can lift objects and morphisms of each $H_{V^i}(\mathcal{C})$ to $H_{V^\omega}(\mathcal{C})$, the closure of $H_{V^\omega}(\mathcal{C})$ under the defined operations is ensured if each operation results in any $H_{V^i}(\mathcal{C})$. This is obivious from the definition of the operations, which use no more than σ, conventional product and conventional co-product.

Theorem 2.6.1 *$H_{V^\omega}(\mathcal{C})$ is closed under non-sharing, sharing and co-products.*

3 Conclusions

As the motivation for defining the sharing product came from Denotational Semantics, we reserve a little space in this section to talk about it. We describe in general lines some of the solutions for sharing and, as parallelism and sharing usually have an operational appeal, we comment some of the differences between Operational and Denotational Semantics, justifying our preference for the latter. We also compare the described solutions and comment how the sharing product shall be used to cope with the problems reported.

3.1 The Denotational and the Operational Approaches

Denotational approach has been widely used as a clear mechanism to give semantics to programming languages. The essence of this approach can be summarised as a way of associating special kinds of objects to syntactical constructions. In this sense, it can be remarked with the following slogan: *denotational approach tells what programs do*. It is usually contrasted with the operational approach in which an abstract machine is considered. Instead of giving the semantic object, this latter emphasises the *construction* of the semantic object using instructions of an abstract machine. The slogan, in this case, would be *operational approach tells how programs work*.

Justifications for the use of one or another approach reflect the essence of both sides of this dichotomy, that is, the conflict $what \times how$. Operational approach tends to be better accepted by programmers. One of the reasons for this is that the underlying machine is usually related to the paradigm of the programming language being considered. For example, an abstract state machine is convenient for giving Operational Semantics for imperative languages, as the latter has *store* and *updating store* as mains concepts, and these notions are very well expressed in terms of *state* and *state changes*. Likewise, application and reduction rules can give rise to an appropriated Operational Semantics to the functional language $\lambda - calculus$[1]. From another point of view, as operational descriptions are closely connected to a machine model, and not to the syntactical parts of the language, they may not reflect the way programmers reason about programs.

3.2 The 'Traditional' Denotational Approach

Sometimes referred to as *the mathematical approach*, Denotational Semantics usually embodies a difficult mathematical framework, what makes it awkward to many users. But all this mathematics is important to capture the complex nature of the objects that syntactical constructions denote. This is the case of

[1] Milner remarks in [Mil89]: *'In fact the lambda − calculus was the first language to be defined by the method of Operational Semantics; the general methodology of Operational Semantics stems from the few rules which define the lambda-calculus'.*

the 'traditional' Denotational Semantics, which is based in the Domain Theory. The following paragraph, in the preface of [VSHG94], puts in few words the general ideas of domains. From this paragraph we can see how the *step by step* computation is enclosed in a mathematical framework, and completely dissociated from the semantic method.

> *'A domain is a structure modelling the notion of approximation and computation. A computation performed using an algorithm proceeds in discrete steps. After each step, there is more information available about the result of the computation. In this way, the result obtained after each step can be seen as an approximation of the final result.'*

A domain models the notion of *data type*, that is, a set of 'things' together with a set of operations that can be applied to them. The 'things' that compose the data type are organised in a domain according to how much defined they are. A program is seen as a function from the data type it inputs to the data type it outputs. Therefore, a *morphism* between domains. At the same time, being itself a function, a program can also be modelled by a domain. To model a function as a domain, we get the graph of this function, that is, set of points of the Cartesian Plan. The empty set of points expresses the undefined function, that is, the point of the execution where we have no information at all about the program. Other sets of points are organised in domains according to the amount of information they give about the program.

This brief intuitive explanation of the domains can give us an insight of how denotational approach encourages descriptions free of implementation aspects: the semantic method plays just the role of linking syntactic and semantic world. Everything else is concerned to an attached theory. In the case of the 'traditional' Denotational Semantics approach, the whole semantics is really free of implementation aspects, as the semantic method is not concerned to describe the semantic world, and the theory used to describe the semantic world is mathematical.

3.3 Essentials of the Denotational Approach

Let us stress the heart of the denotational approach:

- Denotational Semantics is *Compositional*: the mapping between syntax and semantics respects the syntactical structure of the language. In other words, the semantic function must be a homomorphism between the syntactical world and the semantic world.

- Denotational Semantics is *Extensional*: the semantics of a syntactical part is an *object*, something already constructed, that has its meaning by itself. Different denotational approaches will have different frameworks within which these objects are constructed. From this point comes the *what* that characterises the denotational approach: each syntactical part has an object as its denotation.

Now, referring back to section 3.1, the comparison between Denotational and Operational Semantics can be better stated. In opposition to the operational approach, when using Denotational Semantics, programmers would not be induced to adopt decisions persuaded by the abstract machine model[2]. Being free of that concept of *machine*, denotational approach captures the very first idea of programs. At the same time, it gives enough information to verify whether a well-defined function exists for any program[3]. It seems to be suitable for any kind of language, whatever the underlying paradigm is. Thus, despite its usually sophisticated mathematical framework, the 'traditional' Denotational Semantics has played successfully the role of stating clearly and accurately the meaning of programming languages.

3.4 Limitations of the Denotational Approach

There are, however, some features of imperative programming languages that turn out to be very complicated to be expressed within the 'traditional' Denotational Semantics because of their intrinsically operational aspect. In [Mos92], P. Moses argues that the concept of sequential execution becomes too much complicated if expressed in terms of function composition, as usually done in Denotational Semantics.

> *'... there are two main techniques for representing the basic concept of sequential execution: strict composition of functions on states, and reverse composition of functions on continuations. In fact the functions composed typically have further arguments, representing context-dependent information, such as the current bindings; this makes the pattern of composition really quite complex, even for representing something so simple as sequencing.'*

In [Abr97], Abramsky lists what he calls definite limitations of the classical denotational paradigm:

[2]In some situations, however, this can be a wished feature. Although stressing some problems as achieving a rigorous definition of the abstract machine and the need to consider a particular program making it difficult to characterise the well-definedness of *any* program in the language, [Sto89] comments: *'Operational definitions of semantics tend to suggest techniques to compiler writers - this, indeed, may be one of their main benefits'*.

[3]Once again, a Stoy's commentary in [Sto89]: 'To verify that such a well defined function exists for <u>any</u> program in the language is a fundamental task of any mathematical theory of semantics'.

'... fine-structural features of computation, such as sequentiallity, computational complexity and optimality of reduction strategies. [...] Neither concurrency nor "advanced" imperative features such as local references have been captured denotationally in a fully convincing fashion.'

Several approaches were born to cope with the inability of the 'traditional' Denotational Semantics to deal with the features pointed above. Some of them are mainly operational approaches.

In what concerns shared resources, the main problem is that when we pass an argument to a function, the changes in this argument are confined to the internal scope of that function. We can't have an argument shared by two functions in a way that a change within one scope would be reflected in the other function's scope. In this sense, functions are too well behaved stuffs!

The 'traditional' semantics for parallel programs considers that *'[...] in a parallel program the execution of one portion of the program may be interleaved with the execution of another at the end of any of the "indivisible operations"'*[MS76]. The concept of *continuation*, that is, the function that describes the part of the program that is to be executed after a given point, is used to make possible this interleaving. The recursive definition below pictures this technique.

$$R = X \to (X \times R)$$

The domain X can be assumed to be the storage domain, and thus, R represents the sequence of storage changes that will happen after a point.

To form what is nowadays called the *resumption semantics* [Sch86], these ideas are connected with Plotkin's *Power Domains*. This later is the set of all possible results that a function can output. This concept is used to express the inherent non-determinism that arises from parallel programs. Thus, we have

$$R = X \to \mathcal{P}(X \times R)$$

In the above equation the operational 'tasty' is delegated to the recursion, and the solution of the recursive equation will express a kind of 'history' of the program behaviour, as the storage changes are being kept in a n-uple.

What we have, thus, is a way of expressing the parallel behaviour by serialising it: parallel programs are considered a sequence of steps. These steps are interleaved forming a serial execution. As there are several ways of interleaving programs, a power set is used. The effect of shared resources is accomplished by passing the context sensitive information to each of the (serial) possible behaviours in the power set. The final result is: extensionality and compositionallity are preserved, but we get very complicated semantic descriptions. Another point is that the reader loses the feeling of 'things occurring in parallel', as executions are serialised.

Stressing the point of describing programs by its behaviour, Milner [Mil89, Mil19] proposes the *Calculus of Communicating Systems*, or *CCS*. Processes, that is, the programs' behaviours, are the denotation of a program. Processes can be characterised by transducers. Nevertheless, to keep the extensionality of denotational definitions, the transducer itself is not considered, but the set of all processes that it describes.

CCS is an algebra suitable to express parallel processes behaviour. As Milner suggests in [Mil89], we can make an analogy between the role that $\lambda - calculus$ plays with respect to sequential programs and the role that it plays with respect to parallel programs.

As a purely formal system (symbols untied of any meaning and rules for manipulating them), the $\lambda - calculus$ does not help much in giving semantics to programming languages, but, considering an appropriated model [Sto89] (that is, a semantic interpretation for each syntactical element), one can use any consistent translation mechanism from the programming language to the $\lambda - calculus$, and thus, give semantics to the programming language via the $\lambda - calculus$.

Providing a notation powerful enough to express the competition of more than one function for the same argument, *CCS* can express concurrence in a convenient way. So, considering a model for *CCS*, semantics of concurrent programming language can be done via *CCS*. In other words, *CCS* can act as a metalanguage, say, in a denotational approach, in the same way the $\lambda - calculus$ does. It must be stressed, however, that processes are not proved to be domains, so, this approach is not in harmony with the 'traditional' Denotational Semantics.

CCS is not a semantic approach in the sense that it does not attach meaning to syntactical parts of the language, but it is a precise and easy method to describe processes. In [Sch86], Schmidt presents a technique calling it "an alternative semantics for concurrency". It connects programs and their behaviours within a denotational approach: each syntactical expression is mapped in a behaviour expression. Given in this way, semantics is extensional but, in some sense, it fails to be compositional. Extensionality, as already mentioned, comes from the fact that the machine model is not being considered.

To be compositional, however, the semantic approach should have, as background, the following slogan: *give the semantics of the totality as a function of the semantics of its parts*. But when concurrent programs are considered, one may interfere in the behaviour of the other, so semantics can not be done independently. Usually, this context dependence appears in the names which perform communication between the parts and a storage manager.

Another point is that a semantic element (storage manager, or communication manager) is often necessary. But this element is completely dissociated from the syntactical world. Thus, it is not accurate to say that each semantic part comes from a syntactical construction.

We mentioned, thus, two approaches to cope with sharing. The first one is truly Denotational Semantics, because it keeps extensionality and compositionality, but turns out to be very hard to deal with. The second one seems to be easier to deal with, but loses compositionality.

Both approaches have in common the fact of solving the problem in the semantic level. In our opinion, what makes the problem so hard to solve is that the syntactic world is not presented in a way to stress the sharing. As a consequence, the semantic part has to embody an extra elemnt to cope with it. The bridge between syntax and semantics is not evident. The semantic entity that is the denotation of a syntactic construction seems to be completely dissociated from the original syntactical construction.

This paper presented our first ideas of an operator to syntactically express sharing. Our further directions are to adjust this operator to deal with domains, to experiment it in denotational descriptions, and finally, to extend it to other areas of computer science.

4 Appendix

Proposition 4.0.1 *Given two functors $F, G : \mathcal{A} \to \mathcal{B}$, with F faithful, and a natural transformation $\theta : F(\mathcal{A}) \to G(\mathcal{A})$, there is a functor T defined from the image of F in the image of T, in the following way:*

$$\begin{array}{lcccccc} T: & F(\mathcal{A}) & \longrightarrow & G(\mathcal{A}) & & \\ & T(F(a)) & = & cod(\theta_a) & = & G(a) \\ & T(F(f : a \to b)) & = & cod(\theta_a) \to cod(\theta_a) & = & G(f) \end{array}$$

Proof:
First of all, let us show that T act as a mapping. The fact that θ is a natural transformation ensures that each element of the domain of T is mapped in an element of the co-domain. The fact that F is faithful ensures that two components of θ will not have the same domain, thus, each a will be mapped to a unique object in the image of G. The picture below illustrates the necessity of having F faithful. Secondly, we show that T preserves identities and compositions.

$$\begin{tikzcd} \mathcal{A} & & \mathcal{C} & \\ a \arrow[dd] & & & F(a) = F(b) \arrow[dl, "\theta_a"'] \arrow[ddl, "\theta_b"] \\ & & G(a) \arrow[d] & \\ b & & G(b) & \end{tikzcd}$$

(i) T preserves identities: for $a \in Obj(\mathcal{A})$

$$T(Id_{F(a)}) = T(F(Id_a)) = G(Id_a) = Id_{G(a)} = Id_{T(F(a))}$$

(ii) T preserves composition: for $f : a_1 \to a_2, g : a_2 \to a_3 \in Morph(\mathcal{A})$

$$T(F(g)\circ F(f)) = T(F(g\circ f)) = G(g\circ f) = G(g)\circ G(f) = T(F(g))\circ T(F(f))$$

•

Proposition 4.0.2 $Hom_{\mathcal{C}}(a,b) \times Hom_{\mathcal{C}}(a,c) \simeq Hom_{\mathcal{C}}(a, b\times c)$

Proof: *The proof consists in showing an isomorphism α from $Hom_{\mathcal{C}}(a,b) \times Hom_{\mathcal{C}}(a,c)$ to $Hom_{\mathcal{C}}(a,b\times c)$. Thus,*

$$\begin{array}{rccc} \alpha : & Hom_{\mathcal{C}}(a,b)\times Hom_{\mathcal{C}}(a,c) & \longrightarrow & Hom_{\mathcal{C}}(a,b\times c) \\ \alpha' : & Hom_{\mathcal{C}}(a,b\times c) & \longrightarrow & Hom_{\mathcal{C}}(a,b)\times Hom_{\mathcal{C}}(a,c) \end{array}$$

where $\alpha\circ\alpha' = Id_{Hom_{\mathcal{C}}(a,b\times c)}$, and $\alpha'\circ\alpha = Id_{(Hom_{\mathcal{C}}(a,b)\times Hom_{\mathcal{C}}(a,c))}$.
Let π_1 and π_2 be the projections of $a\times b$. Then, α and α' are as follows.

$$\begin{array}{rccc} \alpha : & Hom_{\mathcal{C}}(a,b)\times Hom_{\mathcal{C}}(a,c) & \longrightarrow & Hom_{\mathcal{C}}(a,b\times c) \\ & < f : a\to b, g : a\to c > & \longmapsto & h : a \to b\times c \end{array}$$

such that $\pi_1\circ h = f$ and $\pi_2\circ h = g$, where h is unique by the product properties.

$$\begin{array}{rccc} \alpha' : & Hom_{\mathcal{C}}(a,b\times c) & \longrightarrow & Hom_{\mathcal{C}}(a,b)\times Hom_{\mathcal{C}}(a,c) \\ & h : a\to b\times c & \longmapsto & < h\circ\pi_1, h\circ\pi_2 > \end{array}$$

•

Proposition 4.0.3 $Hom_{\mathcal{C}}(a,b) \times Hom_{\mathcal{C}}(c,b) \simeq Hom_{\mathcal{C}}(a+c, b)$

Proof: *Again, we need an isomorphism from $Hom_{\mathcal{C}}(a,b)\times Hom_{\mathcal{C}}(c,b)$ to $Hom_{\mathcal{C}}(a+c,b)$, Thus,*
Let i_1 and i_2 be the injections of $a+b$. Then, α and α' are as follows.

$$\begin{array}{rccc} \alpha : & Hom_{\mathcal{C}}(a,b)\times Hom_{\mathcal{C}}(c,b) & \longrightarrow & Hom_{\mathcal{C}}(a+c,b) \\ & < f : a\to b, g : a\to c > & \longmapsto & h : a+c \to b \end{array}$$

such that $h\circ\imath_1 = f$ and $h\circ\imath_2 = g$, where h is unique by the co-product properties.

$$\begin{array}{rccc} \alpha' : & Hom_{\mathcal{C}}(a+c,b) & \longrightarrow & Hom_{\mathcal{C}}(a,b)\times Hom_{\mathcal{C}}(c,b) \\ & h : a+c\to b & \longmapsto & < h\circ\imath_1, h\circ\imath_2 > \end{array}$$

•

Proposition 4.0.4 $Hom_{\mathcal{C}}(a,b) \times Hom_{\mathcal{C}}(a,b) \simeq Hom_{\mathcal{C}}(a,b\times b) \simeq Hom_{\mathcal{C}}(a+a,b)$

Proof: *Direct from 4.0.2, with $b = c$, and 4.0.3, with $a = c$*
•

Proposition 4.0.5 $Hom_{\mathcal{C}}(a,b) + Hom_{\mathcal{C}}(a,c) \lesssim Hom_{\mathcal{C}}(a,b+c)$

Proof: *We need a morphism* α *from* $Hom_{\mathcal{C}}(a,b)+Hom_{\mathcal{C}}(a,c)$ *to* $Hom_{\mathcal{C}}(a,b+c)$*, Thus,*
Let i_1 *and* i_2 *be the injections of* $b+c$*. Then,* α *is as follows.*

$$\begin{array}{rccc} \alpha: & Hom_{\mathcal{C}}(a,b) + Hom_{\mathcal{C}}(a,c) & \longrightarrow & Hom_{\mathcal{C}}(a,b+c) \\ & f : a \to b & \longmapsto & \imath_1 \circ f \\ & g : a \to c & \longmapsto & \imath_2 \circ g \end{array}$$

To show that $\gtrsim$ *is not valid, suppose* $a = \{a_1, a_2\}, b = \{b_1\}$ *and* $c = \{c_1\}$*. The function* h *defined below belongs to* $Hom_{\mathcal{C}}(a,b+c)$*, but not to* $Hom_{\mathcal{C}}(a,b)+Hom_{\mathcal{C}}(a,c)$*.*

$$\begin{array}{rccc} h: & a & \to & b+c \\ & a_1 & \longmapsto & b_1 \\ & a_2 & \longmapsto & c_1 \end{array}$$

•

References

[Abr97] Samson Abramsky. Semantics of interaction: an introduction to game semantics. In Andrew M. Pitts and Peter Dybjer, editors, *Semantics and Logics of Computation*. Cambridge University Press, Cambridge, 1997.

[Mil19] Robin Milner. Process: A mathematical model of computing agents. In *Acho que no livro do Stoy tem referencia pra esse paper*. 19.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice hall, New York, 1989.

[Mos92] Peter D. Mosses. *Action Semantics*. Cambridge University Press, 1992.

[MS76] Robert Milne and Christopher Strachey. *A Theory of Programming Language Semantics*. Jonh Wiley and Sons, New York, 1976.

[Sch86] David A. Schmidt. *Denotational Semantics*. Wm. C. Brown Publishers, Bubuque,Iowa, 1986.

[Sto89] Joseph E. Stoy. *Denotational Semantics*. The MIT Press, Cambridge, 1989.

[VSHG94] Ingrid Lindstrom V. Stoltenberg-Hansen and Edward R. Griffor. *Mathematical Theory of Domains*. Cambridge University Press, 1994.