Mathware & Soft Computing 7 (2000) 99-123

Denotational Semantics of Languages with Fuzzy Data

D. Sánchez Alvarez and A.F. Gómez Skarmeta
Depto. de Informática, Inteligencia Artificial y Electrónica
Universidad de Murcia
daniel@dif.um.es

Abstract

The denotational semantics of a programming language which manages fuzzy data is presented. The introduction of blocks poses problems regarding transmission, both for the degree at which the work is carried out and for triangular operations necessary for the evaluation of the degrees of the fuzzy data. We propose some solutions. The possibility of defining linguistic variables is provided.

1 Introduction

A very simple, nondeterministic imperative language is proposed in [5] which we wish to extend in two ways : through the enriching of its structuring and by consideration the needs of the fuzzy calculations. Extension in the case of the former would be useful in order to :

1. Provide it with a blocks structure.

2. Provide it with functions and procedures.

In the case of the latter, it is interesting to :

1. Choose the way of representing the fuzzy sets and the operations defined on them.

2. Be able to define linguistic variables.

To this end we will present in the following sections the extensions and modifications to be performed in the abstracts, control and types. We will give the abstract syntaxis of the language and the different valuation functions. Finally we will give an example to demostrate the language's possibilites.

2 Global Aspects

2.1 Blocks and Abstractions

In the language mentioned there exist two elements necessary for the evaluation of a sentence :

- The fuzzy index, an element of [0,1], hereinafter referred to as Ind_Fuz and
- One of the s functions obtained when defining the lambda calculus_b [4], i.e. a T-norm or an S-conorm. Hereinafter we use T_op to represent the three functions necessary to calculate the belonging values of the intersection and the union of fuzzy sets.

The first was modified in the branches as a consequence of the result of the boolean evaluation of the proof and we were able to use it for fuzzy inference (generalised modus ponens). The second remained somewhat ambiguous and it was never specified whether it was the same T-norm or S-conorm, although our opinion is that it was always the same one. When the blocks are introduced these elements have to be transmitted from one block to another, something which can be performed using different strategies. Something similar occurs with the environment, and the way in which environments are managed in programming languages will serve as a guide to study these strategies.

Virtually all languages include some notion of context. The context in which a sentence is used influences its meaning. In a programming language the contexts are responsible for attributing meaning to the identifiers. In denotational semantics the context of a sentence is modelled by a mechanism known as environment. This concept was not necessary in [5] since the language dealt with there had exactly one environment. This environment was linked to the store, giving rise to an application of the identifiers in the storable values. This very simple model will be divided into two components - the environment and the store.

The environments are used as arguments in the valuation functions. The meaning of a sentence is determined in principle by the function :

$\mathcal{S}$: **Command** $\rightarrow$ **Env** $\rightarrow$ **Ind_Fuz** $\rightarrow$ **T_op** $\rightarrow$ **Sto** $\rightarrow$ $\mathcal{P}$(**Sto**)

such that for a determined index $\in$ **Ind_Fuz** and some determined triangular operators $\in$ **T_op**, the meaning of a sentence is a function **Sto** $\rightarrow$ $\mathcal{P}$(**Sto**), which is determined once the environment establishes the context for the sentence. Thus the environment will belong to the domain

Environment =
Identifiers $\rightarrow$ **Denotable_Values**

where Denotable Values is the domain of all the values that an identifier can represent. For a programme (of a deterministic language) there exists one single store and several environments - those that are necessary to establish the contexts of the different blocks, such as functions and procedures. This means that two possibilities exist when an abstraction is invoked :

- the use of the active environment at the moment the abstraction is defined.
- the use of the active environment at the moment the abstraction is invoked.

2.2 Fuzzy Index and Triangular Operators

The fuzzy index and triangular operator necessary for the evaluation of each sentence are global for each block and, thus, there are several options. For example, in the case of the index, we can consider :

- That the sentences composing the body of the abstraction take as their index the one that exists at the moment of invocation (we could say that it has a dynamic effect).
- That the sentences composing the body of the abstraction take as their index the one that is defined at the moment of the declaration of the abstraction (we could say that it has a static effect)
- Yet a third option exists. That the index for the evaluation of the body of the abstraction is the result of the operating, through a T-norm or S-conorm, the index wich exists at the noment of the invocations with the index that is defined at the moment of the declaration of the abstraction.

Something similar occurs in the case of the triangular operators :

- That the triangular operator is the one used at the moment the abstraction is invoked.
- That the triangular operator is the one established at the moment the abstraction is declared.

2.3 Control

We are going to introduce the sentences **if** E **then** S **el** S **end if** and **while** E **do** S **end do**. The sentences **case** G **esac** and **do** G **od** are a generalisation of both. In order not to lose the orthogonal character of our construction, we will make the sentences **if** E **then** S **else** S **end if** and **while** E **do** S **end do** exclusively control and flow ones. In other words, the degree with which E is evaluated is not transmitted to the index for the evaluation of S.

2.4 Types

We are going to introduce a compact representation for the fuzzy sets. To do so we will use trapezoidal numbers and since we wish to deal with the sets directly, i.e. we want to name them, store them and we want them to be able to be of an operation or of the call to a function, they must form part of the storable values as well as the denotable values and the expressable ones. Furthermore, we want to give the possibility of creating new types, especiañña so as to be able to treat what Zadeh calls linguistic values and linguistic variables [7]. With these meanings are given to colloquial sentences such as barely suitable, suitable, very suitable etc., which are used to refer some characteristics of a specific object. The object, in its simplest form, will be defined from some characteristics observable in determined scales. We can divide each of these scales in a fuzzy way into different sections which we will label with a linguistic value. The universe of the discussion for the object will be the Cartesian product of the characteristics. The fuzzy subsets of the said product, constructed from the logical operators and linguistic values of the characteristics, will the basis from which we will be able to establish linguistic values for the object, see [1] and [2].

For the declaration of these new types we will use Tennet's qualification principle [6]. According to this principle any syntactic domain can have a block in order to admit local declarations. We will apply this specifically to the extension of the "records". As the body of a record is a declaration we are going to allow functions to appear in it also. This is semantically correct, since each record is a species of environment in which each identifier is linked to a denotable value and it can, therefore, be a function. Thus we will obtain a kind of *class*. Furthermore, if to this structure we add a list of pairs of identifiers, we will be able to define antinomes.

3 The language

Next we present the components and syntax of our language

3.1 Abstract syntax

P $\in$ Programs

K $\in$ Block

D $\in$ Declarations

D_c $\in$ Constant definition

D_t $\in$ Type definition

D_v $\in$ Variable declaration

S $\in$ Commands

E $\in$ Expressions

G $\in$ Guarded commands

T $\in$ Types

I $\in$ Identifiers

Ω $\in$ Dyadic operators

Υ $\in$ Monadic operators

Π $\in$ Parameters

V_l $\in$ Linguistic value

G_r $\in$ Degree

$T_o \in$ T_op

N $\in$ Numerals

B $\in$ Boolean

3.2 Grammar

P ::= K.

K ::= D **begin** S **end**

D ::= **const** D_c^*
  | **var** D_v^*
  | **type** D_t^*
  | **function** I (Π^*) T ; G_r T_o K
  | **procedure** I (Π^*) ; G_r T_o K

D_c ::= I = E

D_v ::= I T

D_t ::= I = **li_ty** Π^* **li_va** $(V_l)^*$ **anti** $(\text{I I})^*$ **end**

Π ::= I T

T ::= **integer** | **boolean** | **real** | **c_fuzzy** | **fuzzy** | I

V_l ::= I (Π^*) T ; G_r T_o K
  | **as** I

S ::= I **:=** E
  | I(E^*)
  | **if** E **then** S **else** S **end if**
  | **while** E **do** S **end do**
  | **print**(E^*)
  | **case** G **esac**
  | **do** G **od**
  | K
  | **skip**
  | **return**
  | I_1 <- I_2
  | S ; S

G ::= E -> S | G $\Box$ G

E ::= I | N | B | E Ω E | Υ E | I(E^*) | I.E

3.3 Semantic algebras

As we indicated above, we are going to describe the dynamic denotational semantics. In this semantics the assignation of a symbol to represent errors is irrelevant since we suppose that the programs which contain them have been rejected as illegal. Thus we reserve $\perp$ to represent the *non-termination*, and no symbol will be introduced to represent errors in all the domains and we will not even specify the treatment of such errors.

NOTE.- From here on, the notation given in [3] and is used for semantic domains. Furthermore, since we make use of the "lambda-calculus_b" [4], we must consider that each time we talk about functions, these will always have to carry a degree, and in the case of the degree being the unit n of $\mathbb{D}$, we will omit it. For example, in the elements $\rho \in \Sigma$, we must consider it as (ρ, n). The index that we have introduced to reflect the degree with which we are working, since it is not an element of the "lambda-calculus_b", we substitute for a family of functions $I = (\lambda x.x, q)$ where $q \in \mathbb{D}$.

I LITERALS

(LITERAL)
L ::= B | N | R | G | C | CC

(BOOLEAN)
B ::= true | false

(NUMERAL)
N ::= *unspecified*

(NUMERAL REAL)
R ::= *unspecified*

(DEGREE)
Grd ::= *unspecified*

(CHARACTER)
C ::= *unspecified*

(CHARACTER-STRINGS)
CS ::= *unspecified*

(a) Domain **Bool** = **T**
Operations
- **true, false**: **Bool**
- **or, and**: (**Bool** $\otimes$ **Bool**) $\circ\!\!\rightarrow$ **Bool**
- **not**: **Bool** $\circ\!\!\rightarrow$ **Bool**

(b) Domain **Num** = *unspecified*
Operations
- **zero, one, ...**: **Num**
- **add, minus, times, div**: (**Num** $\otimes$ **Num**) $\circ\!\!\rightarrow$ **Num**

(c) Domain **Grd** = $[0, 1]$
Operations

- zerog, oneg, zerofiveg, ...: **Grd**
- Operador_triangular:
  $(\mathbf{Grd} \otimes \mathbf{Grd}) \rightarrow \mathbf{Grd}$
  – *According to strategy*

(d) Domain **Real** = *unspecified*
Operations

- **zeror, oner,** ...: **Real**
- **addr, minusr, timesr. divr**: $(\mathbf{Real} \otimes \mathbf{Real}) \circ\!\!\rightarrow \mathbf{Real}$

(e) Domain **Char** = *unspecified*
Operations

- **ord**: $\mathbf{Char} \circ\!\!\rightarrow \mathbf{Num}$
- **chr**: $\mathbf{Num} \circ\!\!\rightarrow \mathbf{Char}$

(f) Domain **String** = *unspecified*
Operations

- **str**: $\mathbf{Char}^* \circ\!\!\rightarrow \mathbf{String}$
- **chrs**: $\mathbf{String} \circ\!\!\rightarrow \mathbf{Char}^*$

II IDENTIFIERS
It is supposed that there exists a flat domain **Ide** that corresponds to a class called IDENTIFIERS.

III FUZZY BOOLEANS
Domain $\mathbb{B}_{\mathbb{G}} = \mathbf{Bool} \otimes \mathbf{Grd}$

IV FUZZY NUMBERS
Domain $\mathbb{N}_{\mathbb{G}} = \mathbf{Num} \otimes \mathbf{Grd}$

V FUZZY REALS
Domain $\mathbf{RealB} = \mathbf{Real} \times \mathbf{Grd}$

VI TRAPEZOIDAL NUMBERS
Domain $\mathbf{NumT} = \mathbf{Real} \times \mathbf{Real} \times \mathbf{Real} \times \mathbf{Real} \times \mathbf{Grd}$
Operations

- addt, minust, multt, divt : $\mathbf{NumT} \otimes \mathbf{NumT} \circ\!\!\rightarrow \mathbf{NumT}$

VII FUZZY SETS
Domain $\mathbf{C\_fuzzy} = \mathcal{P}(\mathbf{Num} \times \mathbf{Grd})$
Operations

- union, intersection: $\mathbf{C\_fuzzy} \otimes \mathbf{C\_fuzzy} \circ\!\!\rightarrow \mathbf{C\_fuzzy}$

VIII STORAGE LOCATIONS
Domain **Loc**
Operations

- first_locn: **Loc**
  - **first_locn**= *Parameter*
- next_locn: **Loc** → **Loc**
  - **next_locn**= *unspecified*
- equal_locn, lessthan_locn: **Loc** × **Loc** → **T**

IX EXPRESSABLE VALUES
We want **NumT** to be "first category", i.e. that it can passed as parameter to a function or be sent back by it etc. Thus **EV**, el domain of the result of the evaluation of expressions, becomes:

Domain **EV** =
BoolB ⊕ **NumB** ⊕ **RealB** ⊕ **NumT** ⊕ **C_fuzzy**

X DENOTABLE VALUES
We enrich our language by allowing the existence of constans, functios, types, etc. Thus the set of values that can be "denoted" by identifiers takes following form:

Domain **DV** =
Const ⊕ **Loc** ⊕ **Abst** ⊕ **VaL** ⊕ **TiL**
where
Const = **EV**
and
Abst = **Func** ⊕ **Proc**

XI FUNCTIONS

$$\mathbf{Func} = \mathbf{Param} \circ\!\!\rightarrow \mathbf{Ind\_Fuz} \circ\!\!\rightarrow \mathbf{T\_op} \circ\!\!\rightarrow (\mathbf{Sto} \otimes \mathbf{Out}) \circ\!\!\rightarrow (\mathbf{EV} \times ((\mathbf{Sto} \otimes \mathbf{Out})_\perp \oplus \delta))^\natural$$

XII PROCEDURES

$$\mathbf{Proc} = \mathbf{Param} \circ\!\!\rightarrow \mathbf{Ind\_Fuz} \circ\!\!\rightarrow \mathbf{T\_op} \circ\!\!\rightarrow (\mathbf{Sto} \otimes \mathbf{Out}) \circ\!\!\rightarrow ((\mathbf{Sto} \otimes \mathbf{Out})_\perp \oplus \delta)^\natural$$

where **Param** = **EV**

XIII LINGUISTIC VARIABLES

$$\mathbf{VaL} = \mathbf{Ide} \rightarrow (\mathbf{Loc} \oplus \mathbf{Func} \oplus \mathbf{VaL})$$

XIV TYPES

$$\mathbf{TiL} = \mathbf{Env} \circ\!\!\rightarrow \mathbf{Sto} \circ\!\!\rightarrow (\mathbf{DV} \times \mathbf{Sto})$$

XV Environments

On giving our language a block structure, we require environments in order to represent the associations between the identifiers and the denoted values. The element $\top \in$**O** is used to indicate the absence of the denoted value.

Domain **Env** = **Ide** → (**DV** ⊕ **O**)

Operations

- empty_env: **Env**

  empty_env = $\lambda \mathrm{id}_{\in \mathbf{Ide}}.\ \mathbf{in}_2 \top$

- bound: **Ide** → **Env** → **DV**

  bound = $\lambda \mathrm{i}_{\in \mathbf{Ide}}.\lambda \mathrm{a}_{\in \mathbf{Env}}.[\ \mathbf{id}_{\mathbf{DV}},\ \perp\ ](\mathrm{a}(\mathrm{i}))$

- binding: **Ide** → **DV** → **Env**

  binding = $\lambda \mathrm{i}_{\in \mathbf{Ide}}.\lambda \mathrm{v}_{\in \mathbf{DV}}.\lambda \mathrm{i}'_{\in \mathbf{Ide}}.$
  if $\mathrm{i} =_{\mathbf{Ide}} \mathrm{i}'$ then $\mathbf{in}_1(\mathrm{v})$ else $\mathbf{in}_2(\top)$

- overlay: **Env** × **Env** → **Env**

  overlay = $\lambda(\mathrm{a}_{\in \mathbf{Env}},\mathrm{a}'_{\in \mathbf{Env}}).\lambda \mathrm{i}_{\in \mathbf{Ide}}.$
  $[\ \mathbf{id}_{\mathbf{DV}},\ \lambda \mathrm{x}_{\in \mathbf{O}}.\ \mathrm{a}'(\mathrm{i})\ ](\mathrm{a}(\mathrm{i}))$

- update_env: **Ide** → **DV** → **Env** → **Env**

  update_env = $\lambda \mathrm{i}_{\in \mathbf{Ide}}.\lambda \mathrm{d}_{\in \mathbf{DV}}.\lambda \mathrm{a}_{\in \mathbf{Env}}.$
  overlay a (**binding** i d)

- combine: **Env** × **Env** → **Env**

  combine = $\lambda(\mathrm{a}_{\in \mathbf{Env}},\mathrm{a}'_{\in \mathbf{Env}}).$
  $\lambda \mathrm{i}_{\in \mathbf{Ide}}.\ [\lambda \mathrm{v}_{\in \mathbf{DV}}.[\lambda \mathrm{x}_{\in \mathbf{O}}.\mathrm{v},\perp],\lambda \mathrm{x}_{\in \mathbf{O}}.\ \mathbf{id}_{\mathbf{DV}\oplus\mathbf{O}}]$
  $(\mathrm{a}(\mathrm{i}))(\mathrm{a}'(\mathrm{i}))$

XVI Storable values

The Domain **SV** = **EV** is used in order to represent the set of values that can be stored in a single location.

XVII Store: Memory based on a stack

When administering the stores it is only necessary to know whether a location is reserved or not, which means, for example, that the function asig_loc is left unspecified and the whole model is simplified.

Domain **Sto** = **Loc** → (**SV** ⊕ **O**) × **Loc**

Operations

- empty_sto: **Sto**

  empty_sto = $(\lambda \mathrm{l}_{\in \mathbf{Loc}}.\mathbf{in}_2(\top),\mathrm{first\_locn})$

- access_sto: **Loc** → **Sto** → **SV**

  access_sto =
  $\lambda \mathrm{l}_{\in \mathbf{Loc}}.\lambda(\mathrm{map}_{\in \mathbf{Loc}\to(\mathbf{SV}\oplus\mathbf{O})},\mathrm{l}'_{\in \mathbf{Loc}}).$
  if l lessthan_locn l′ then (map(l))
  else $\mathbf{in}_2(\top)$

· update_sto: **Loc** $\rightarrow$ **SV** $\rightarrow$ **Sto** $\rightarrow$ **Sto**

update_sto =
$\lambda l_{\in \mathbf{Loc}}.\lambda v_{\in \mathbf{SV}}.\lambda(map_{\in \mathbf{Loc} \rightarrow (\mathbf{SV} \oplus \mathbf{O})}, l'_{\in \mathbf{Loc}})$.
if l lessthan_locn l′ then
($\lambda l''$. if $l =_{\mathbf{Loc}} l''$ then v
else ((map,l′)(l″)),l′)
else ($\lambda l_{\in \mathbf{Loc}}.\mathbf{in}_2(\top)$,l′)

· mark_loc: **Sto** $\rightarrow$ **Loc**

mark_loc = $\lambda(map_{\in \mathbf{Loc} \rightarrow (\mathbf{SV} \oplus \mathbf{O})}, l_{\in \mathbf{Loc}}).l$

· al_loc: **Sto** $\rightarrow$ **Loc** $\times$ **Sto**

al_loc = $\lambda(map_{\in \mathbf{Loc} \rightarrow (\mathbf{SV} \oplus \mathbf{O})}, l \in \mathbf{Loc})$.
(l, (map, next_locn(l)))

· deal_loc: **Loc** $\rightarrow$ **Sto** $\rightarrow$ **Sto**

deal_loc =
$\lambda l_{\in \mathbf{Loc}}.\lambda(map_{\in \mathbf{Loc} \rightarrow (\mathbf{SV} \oplus \mathbf{O})}, l'_{\in \mathbf{Loc}}).(map,l)$

XVIII OUTPUT

Domain **Out** = (**SV** $\oplus$ **String**)*

Operations

· empty_out: **Out**

empty_out = $\emptyset$

· put_val: ((**Const** $\oplus$ **String**) $\times$ **Out**) $\rightarrow$ **Out**

put_val = $\lambda(v_{\in \mathbf{Const} \oplus \mathbf{String}}, s_{\in \mathbf{Out}})$. s::v

3.4 Valuation functions

3.4.1 Program and blocks

In the classical languages the programs to be executed need only one parameter, the first direction of the memory that they can use. Our language also requires that when a program is invoked, what we have globally called strategy be passed to it as parameter. This would be made up of:

- The t-norm or conorm chosen to be used instead of the s operation of the lambda-calculus_b.
- The order to be considered in Grd.
- The unit $\in$ Grd of s.
- The initial value at which the evaluations are to be made.

It is supposed that points 2 and 3 are coherent with point 1 and that the initial value is the unit one of the selected norm. Thus, we propose to introduce the following parameters :

1. **Ind_Fuz**. This used at the moment of storing any value in the store. It will alter as result of the execution of a stored command or the execution of a function or a procedure. This alteration will only affect the body of the function, procedure or command stored
2. **T_op**. This is used at the moment of evaluating expressions. It will consist of the triangular operators necessary for the union and intersection of fuzzy sets.

$\mathcal{P}$: PROGRAM $\rightarrow$ **Env** $\rightarrow$ **Ind_Fuz** $\rightarrow$ **T_op** $\rightarrow$
$((\mathbf{Sto} \otimes \mathbf{Out})_\perp \oplus \delta)^\natural$

$\mathcal{P}[[\mathrm{K.}]] = \lambda e_{\in \mathbf{Env}}.\lambda g_{\in \mathbf{Ind\_Fuz}} \lambda t_{\in \mathbf{T\_op}}.\mathcal{K}[[\mathrm{K}]]$

$\mathcal{K}$: BLOCK $\rightarrow$ **Env** $\rightarrow$ **Ind_Fuz** $\rightarrow$ **T_op** $\rightarrow$
$(\mathbf{Sto} \otimes \mathbf{Out}) \rightarrow ((\mathbf{Sto} \otimes \mathbf{Out})_\perp \oplus \delta)^\natural$

$\mathcal{K}[[\text{ D } \mathbf{begin} \text{ S } \mathbf{end}]] =$
$\lambda e_{\in \mathbf{Env}}.\lambda i_{\in \mathbf{Ind\_Fuz}}.\lambda t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto} \otimes \mathbf{Out}}.$
$\underline{\lambda} l_{\in \mathbf{Loc}}.\underline{\lambda}(e_{1 \in \mathbf{Env}}, s_{1 \in \mathbf{Sto}}).\underline{\lambda} a_{2 \in \mathbf{Sto} \otimes \mathbf{Out}}.$
$\mathbf{smash}((\mathbf{deal\_loc}\ l)^\natural\ \mathbf{on}_1 a_2,\ \mathbf{on}_2 a_2))$
$\mathcal{S}[[\mathrm{S}]]\ e_1\ i\ t\ \mathbf{smash}(s_1, \mathbf{on}_2 a))$
$(\mathbf{on}_1(\mathcal{D}[[\mathrm{D}]] e\ \mathbf{on}_1 a)$
$\mathbf{on}_2(\mathcal{D}[[\mathrm{D}]] e\ \mathbf{on}_1 a))\ \mathbf{mar\_loc}\ \mathbf{on}_1 a$

3.4.2 Declarations

In this section we define the classical evaluation functions for the declarations in any imperative language.

$\mathcal{D}$: DECLARATION $\rightarrow$ **Env** $\rightarrow$ **Sto** $\rightarrow$
Ind_Fuz $\rightarrow$ **T_op** $\rightarrow$(**Env** $\times$ **Sto)**

$\mathcal{D}[[\ \mathbf{const}\ \mathrm{I} = \mathrm{E}\ ]] =$
$\lambda e_{\in \mathbf{Env}}.\lambda a_{\in \mathbf{Sto}}.\lambda g_{\in \mathbf{Ind\_Fuz}} \lambda t_{\in \mathbf{T\_op}}$
$\underline{\lambda}(d_{\in \mathbf{EV}}, a_{1 \in \mathbf{Sto}}).$
$((\mathbf{update\_env}\ \mathrm{I}\ \mathbf{in}_1^{\mathbf{DV}}(d)\ e), a_1)\mathcal{E}[[\mathrm{E}]] e\ a$

$\mathcal{D}[[\ \mathbf{var}\ \mathrm{I} : \mathrm{T}\ ]] =$
$\lambda e_{\in \mathbf{Env}}.\lambda a_{\in \mathbf{Sto}}.\lambda i_{\in \mathbf{Ind\_Fuz}}.\lambda t_{\in \mathbf{T\_op}}.$
$\underline{\lambda}(d_{\in \mathbf{DV}}, a_{1 \in \mathbf{Sto}}).$
$((\mathbf{update\_env}\ \mathrm{I}\ d\ e),\ a_1)\ \mathcal{T}[[\mathrm{T}]]\ e\ a$

$\mathcal{D}[[\ \mathbf{procedure}\ \mathrm{I}\ (\Pi^*);\ \mathrm{K}\ g\ ]] =$
$\lambda e_{\in \mathbf{Env}}.\lambda a_{\in \mathbf{Sto}}.\lambda i_{\in \mathbf{Ind\_Fuz}}.\lambda t_{\in \mathbf{T\_op}}$
$((\mathbf{update\_env}\ \mathrm{I}\ \mathbf{in}_3^{\mathbf{DV}}(\mathbf{in}_2^{\mathbf{Abst}}(\lambda d^*_{\in \mathbf{EV}^*}.$
$\lambda c_{\in \mathbf{Sto} \otimes \mathbf{Out}}.\ (\lambda(e_{1 \in \mathbf{Env}}, c_{1 \in \mathbf{Sto} \otimes \mathbf{Out}}).$
$(\lambda a_{2 \in \mathbf{Sto} \otimes \mathbf{Out}}.\lambda i_{\in \mathbf{Ind\_Fuz}}.\mathcal{K}[[\mathrm{K}]]\ e_1\ (t(i,g),t)\ a_2)$
$(\mathcal{R}^*[[\ \mathrm{I}^*\ ]]\ d^*)\ e_1\ c_1)\mathcal{Q}^*\ [[\ \Pi^*\ ]]\ e\ c))\ e),\ a)$

where

$\mathcal{Q}^*$:PARAMETERS → **Env** → **Sto** →
(**Env** × **Sto**)

$\mathcal{Q}^*$ [[Π^*]] = $\lambda e_{\in \mathbf{Env}}.\lambda a_{\in \mathbf{Sto}}.$
$\lambda(e_{1 \in \mathbf{Env}}, a_{1 \in \mathbf{Sto}}) \ldots \lambda(e_{n \in \mathbf{Env}}, a_{n \in \mathbf{Sto}}).$
(... (($\mathcal{D}$[[I_n: T_n]])$\mathcal{D}$[[I_{n-1}: T_{n-1}]]) ...
$\mathcal{D}$[[I_1: T_1]])

and

$\mathcal{R}^*$:PARAMETERS × EXPRESSIBLE VALUES
→ **Env** → **Sto** → **Sto**

$\mathcal{R}^*$ [[I^*]] d^* = $\lambda e_{\in \mathbf{Env}}.\lambda a_{\in \mathbf{Sto}}.$
$\lambda a_{1 \in \mathbf{Sto}} \ldots \lambda a_{n \in \mathbf{Sto}}.$
(... (($\mathcal{R}$[[I_n]] d_n e a_n)
$\mathcal{R}$[[I_{n-1} d_{n-1} e a_{n-1}]]) ... $\mathcal{R}$[[I_1 d_1 e a_1]])

where

$\mathcal{R}$:PARAMETER × VALOR EXPRESABLE
→ **Env** → **Sto** → **Sto**

$\mathcal{R}$ [[I]] d = $\lambda e_{\in \mathbf{Env}}.\lambda a_{\in \mathbf{Sto}}.$
[⊤,[$\lambda l_{\in \mathbf{Loc}}$ **if** $\mathcal{T}'$(l) = $\mathcal{T}'$(d)
then mod_alm l d a
else ⊤],⊤,⊤,⊤] (**bound** I e)

$\mathcal{D}$[[**function** I (Π^*): T; K g]] =
$\lambda e_{\in \mathbf{Env}}.\lambda a_{\in \mathbf{Sto}}.\lambda i_{\in \mathbf{Ind\_Fuz}}.\lambda t_{\in \mathbf{T\_op}}$
((**update_env** I $\mathbf{in}_3^{\mathbf{DV}}$($\mathbf{in}_1^{\mathbf{Abst}}$
($\lambda d^*_{\in \mathbf{EV}}.\lambda c_{\in \mathbf{Sto} \otimes \mathbf{Out}}.(\lambda(e_{1 \in \mathbf{Env}}, c_{1 \in \mathbf{Sto} \otimes \mathbf{Out}}).$
($\lambda a_{2 \in \mathbf{Sto} \otimes \mathbf{Out}}.\lambda.i_{\in \mathbf{Ind\_Fuz}}.$
$\mathcal{E}$ [[_I]] e_1 ($\mathcal{K}$[[K]] e_1 (t(i,g),t) a_2))
($\mathcal{R}^*$[[I^*]] d^*) e_1 c_1)$\mathcal{Q}^*$ [[Π^*::(_I:T)]] e c))e),a)

$\mathcal{D}$[[**type** I = **li_ty** D]] = $\lambda e_{\in \mathbf{Env}}.\lambda a_{\in \mathbf{Sto}}.$
((**update_env** I $\mathbf{in}_5^{\mathbf{DV}}$
($\lambda a_{1 \in \mathbf{Sto}}.$($\mathcal{T}$[[**til** D]] e a_1)) e), a)

$\mathcal{D}$[[D_1; D_2]] = $\lambda e_{\in \mathbf{Env}}.\lambda a_{\in \mathbf{Sto}}.$
$\mathcal{D}$[[D_2]] $\mathbf{on}_1$($\mathcal{D}$[[D_1]] e a) $\mathbf{on}_2$($\mathcal{D}$[[D_1]] e a)

3.4.3 Types

In this section, as well as classical evaluation functions for the types in any imperative language, we define the evaluation functions of the linguistic types.

$\mathcal{T}$: TYPES → **Env** → **Sto** → (**DV** × **Sto**)

$\mathcal{T}[[\ \textbf{boolean}\ ]] = \lambda e_{\in \mathbf{Env}}.\lambda a_{\in \mathbf{Sto}}.$
$\underline{\lambda}(l_{\in \mathbf{Loc}}, a_{1 \in \mathbf{Sto}}).(\mathbf{in}_2^{\mathbf{DV}}(\mathbf{in}_1^{\mathbf{Loc}}(l)), a_1)\ \mathbf{al\_loc}\ a$

$\mathcal{T}[[\ \textbf{integer}\ ]] = \lambda e_{\in \mathbf{Env}}.\lambda a_{\in \mathbf{Sto}}.$
$\underline{\lambda}(l_{\in \mathbf{Loc}}, a_{1 \in \mathbf{Sto}}).(\mathbf{in}_2^{\mathbf{DV}}(\mathbf{in}_2^{\mathbf{Loc}}(l)), a_1)\ \mathbf{al\_loc}\ a$

$\mathcal{T}[[\ \textbf{real}\ ]] = \lambda e_{\in \mathbf{Env}}.\lambda a_{\in \mathbf{Sto}}.$
$\underline{\lambda}(l_{\in \mathbf{Loc}}, a_{1 \in \mathbf{Sto}}).(\mathbf{in}_2^{\mathbf{DV}}(\mathbf{in}_3^{\mathbf{Loc}}(l)), a_1)\ \mathbf{al\_loc}\ a$

$\mathcal{T}[[\ \textbf{borroso}\ ]] = \lambda e_{\in \mathbf{Env}}.\lambda a_{\in \mathbf{Sto}}.$
$\underline{\lambda}(l_{\in \mathbf{Loc}}, a_{1 \in \mathbf{Sto}}).(\mathbf{in}_2^{\mathbf{DV}}(\mathbf{in}_4^{\mathbf{Loc}}(l)), a_1)\ \mathbf{al\_loc}\ a$

$\mathcal{T}[[\ \textbf{c\_fuzzy}\ ]] = \lambda e_{\in \mathbf{Env}}.\lambda a_{\in \mathbf{Sto}}.$
$\underline{\lambda}(l_{\in \mathbf{Loc}}, a_{1 \in \mathbf{Sto}}).(\mathbf{in}_2^{\mathbf{DV}}(\mathbf{in}_5^{\mathbf{Loc}}(l)), a_1)\ \mathbf{al\_loc}\ a$

$\mathcal{T}[[\ \textbf{TiL}\ D\ ]] = \lambda e_{\in \mathbf{Env}}.\lambda a_{\in \mathbf{Sto}}.$
$\underline{\lambda}(e_{1 \in \mathbf{Env}}, a_{1 \in \mathbf{Sto}}).(\mathbf{in}_4^{\mathbf{DV}}(e_1), a_1)\ \mathcal{D}[[\ D\ ]]\ e\ a$

$\mathcal{T}[[\ I\ ]] = \lambda e_{\in \mathbf{Env}}.\lambda a_{\in \mathbf{Sto}}.$
$[\top, \top, \top, \top, \lambda f_{\in \mathbf{Sto} \to (\mathbf{DV} \times \mathbf{Env})}.f\ a]\mathbf{acc\_env}\ I\ e$

$\mathcal{T}'$: TYPES $\to \mathbf{Env} \to (\mathbf{EV} \oplus \mathbf{O})$

$\mathcal{T}'[[\ I\ ]] = \lambda e_{\in \mathbf{Env}}.$
$[\top, \lambda l_{\in \mathbf{Loc}}.\mathbf{id}_{\mathbf{EV}}, \top, \top, \top]\ \mathbf{acc\_env}\ I\ e$

3.4.4 Sentences

As we have already indicated for the programmes the evaluation functions for the sentences require, besides the classical store and environment, what we have called Ind_fuz and T_op. Neither will be modified during the execution of the sentences. The consequence of the execution of a sentence will be a subset of the stores. This subset will be formed by a single store except in the case of the sentence being a guarded command.

$\mathcal{S}$: COMMAND $\to \mathbf{Env} \to \mathbf{Ind\_Fuz} \to \mathbf{T\_op}$
$\to (\mathbf{Sto} \otimes \mathbf{Out}) \to ((\mathbf{Sto} \otimes \mathbf{Out})_\perp \oplus \delta)^\natural$

$\mathcal{S}[[\ I := E\ ]] =$
$\lambda e_{\in \mathbf{Env}}.\lambda i_{\in \mathbf{Ind\_Fuz}}.\lambda t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto} \otimes \mathbf{Out}}.$
$\underline{\lambda}(v_{\in \mathbf{EV}}, a_{1 \in \mathbf{Sto} \otimes \mathbf{Out}}).\lambda l_{\in \mathbf{Loc}}.$
$((\mathbf{update\_sto}\ l\ i_2(v, i_1)\ \mathbf{on}_1\ a_1), \mathbf{on}_2\ a_1)^\natural$
$(\mathbf{bound}\ [[I]]\ e)(\mathcal{E}[[E]]\ e\ t\ a)$

$\mathcal{S}[[\ \textbf{if}\ E\ \textbf{then}\ S\ \textbf{end if}\ ]] =$
$\lambda e_{\in \mathbf{Env}}.\lambda i_{\in \mathbf{Ind\_Fuz}}.\lambda t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto} \otimes \mathbf{Out}}.$
$(\underline{\lambda}(t_{1 \in \mathbf{Bool}}, i_{t \in \mathbf{Grd}}).\ \textbf{if}\ t_1\ \textbf{then}\ \mathcal{S}[[S]]\ e\ i\ t\ a$
$\textbf{else}\ \{\!|\ a\ |\!\})(\mathcal{B}[[E]]\ e\ t\ a)$

$\mathcal{S}[[\ \textbf{while}\ E\ \textbf{do}\ S\ \textbf{end do}\ ]] =$
$\lambda e_{\in \mathbf{Env}}.\lambda i_{\in \mathbf{Ind\_Fuz}}.\lambda t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto} \otimes \mathbf{Out}}.$
$\mathbf{fix}(\lambda f_{\in \mathbf{Sto} \to (\mathbf{Sto} \otimes \mathbf{Out})^\natural}.\lambda a_{\in \mathbf{Sto}}.\lambda(t_{1 \in \mathbf{Bool}}, i_{1 \in \mathbf{Grd}}).$

if t_1 **then ext**(f)($\mathcal{S}$[[S]] e i t a)
else $\{\!|$ a $|\!\}$)($\mathcal{B}$[[E]] e t a)

$\mathcal{S}$[[I(E*)]] =
$\underline{\lambda} e_{\in \mathbf{Env}}.\lambda i_{\in \mathbf{Ind\_Fuz}}.\lambda t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto} \otimes \mathbf{Out}}.$
[⊤,⊤,[⊤,$\lambda p_{\in \mathbf{Proc}}$.(p ($\mathcal{E}$[[E*]] e t a) i t a)],⊤,⊤]
(**bound**[[I]] e a)

$\mathcal{S}$[[**case** G **esac**]] =
$\lambda e_{\in \mathbf{Env}}.\lambda i_{\in \mathbf{Ind\_Fuz}}.\lambda t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto} \otimes \mathbf{Out}}.$
($\underline{\lambda}$($t_{1 \in \mathbf{Bool}}$,$i_{1 \in \mathbf{Grd}}$).**if** t_1 **then** $\mathcal{G}$[[G]] e i t a
else $\{\!|$ δ $|\!\}$)) $\mathcal{V}$[[G]] e a

$\mathcal{S}$[[**do** G **od**]] =
$\lambda e_{\in \mathbf{Env}}.\lambda i_{1 \in \mathbf{Ind\_Fuz}}.\lambda t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto} \otimes \mathbf{Out}}.$
fix($\underline{\lambda}$($t_{1 \in \mathbf{Bool}}$,$i_{1 \in \mathbf{Grd}}$).$\lambda f_{\in \mathbf{Sto} \rightarrow (\mathbf{Sto} \otimes \mathbf{Out})^{\natural}}$.
$\lambda a_{\in \mathbf{Sto} \otimes \mathbf{Out}}.$
if t_1 **then ext**(f)($\mathcal{G}$[[G]] e i t a)
else $\{\!|$ a $|\!\}$)($\mathcal{V}$[[G]] e a)

$\mathcal{S}$[[K]] =
$\lambda e_{\in \mathbf{Env}}.\lambda i_{\in \mathbf{Ind\_Fuz}}.\lambda t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto} \otimes \mathbf{Out}}.$
$\mathcal{K}$[[K]] e i t a

$\mathcal{S}$[[S_1; S_2]] =
$\lambda e_{\in \mathbf{Env}}.\lambda i_{\in \mathbf{Ind\_Fuz}}.\lambda t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto} \otimes \mathbf{Out}}.$
ext($\mathcal{S}$[[S_2]] e i t) $\mathcal{S}$[[S_1]] e i t a

$\mathcal{S}$[[**skip**]] =
$\lambda e_{\in \mathbf{Env}}.\lambda i_{\in \mathbf{Ind\_Fuz}}.\lambda t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto} \otimes \mathbf{Out}}.\{\!|$ a $|\!\}$

$\mathcal{S}$[[**return**]] =
$\lambda e_{\in \mathbf{Env}}.\lambda i_{\in \mathbf{Ind\_Fuz}}.\lambda t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto} \otimes \mathbf{Out}}.\{\!|$ a $|\!\}$

$\mathcal{S}$[[I_1 <- I_2]] =
$\lambda e_{\in \mathbf{Env}}.\lambda i_{\in \mathbf{Ind\_Fuz}}.\lambda t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto} \otimes \mathbf{Out}}.\{\!|$ a $|\!\}$

$\mathcal{S}$[[**print** E]] =
$\lambda e_{\in \mathbf{Env}}.\lambda i_{\in \mathbf{Ind\_Fuz}}.\lambda t_{\in \mathbf{T\_op}}.\lambda(a_{\in \mathbf{Sto}},s_{\in \mathbf{Out}}).$
$\{\!|$(a,**put_val** $\mathcal{E}$([[E]] e t a) s) $|\!\}$

3.4.5 Guarded commands

As indicated in [5], due to the intrinsic paralelism of the language we need the use of guarded command. For it we define two valuation functions $\mathcal{V}$ and $\mathcal{G}$. The first decides whether any branching exists which must be followed, the second executes the sentences associated to the cases whose test is true. The Ind_fuz. which will affect these sentences will be the consequence of the global Ind_fuz., the degree of evaluation of the test and the T_op. As we have already indicated, the exit of the guarded commands may be a multiple one.

$\mathcal{V}$: GUARDED COMMAND $\rightarrow$ **Env** $\rightarrow$ **Sto** $\rightarrow$ **BoolB**

$\mathcal{V}[[\ G_1 \ \Box\ G_2\ ]] = \lambda e_{\in \mathbf{Env}} \lambda a_{\in \mathbf{Sto}}.$
$(\mathcal{V}[[G_1]]\ e\ a)\ \mathit{or}\ (\mathcal{V}[[G_2]]\ e\ a)$

$\mathcal{V}[[\ E \rightarrow S\ ]] = \lambda e_{\in \mathbf{Env}}.\lambda a_{\in \mathbf{Sto}}.\ \mathcal{B}[[E]]\ e\ a$

$\mathcal{G}$: GUARDED COMMAND $\rightarrow$ **Env** $\rightarrow$ **Ind_Fuz** $\rightarrow$ **T_op** $\rightarrow$ (**Sto** $\otimes$ **Out**)$\rightarrow$ $((\mathbf{Sto} \otimes \mathbf{Out})_\perp \oplus \delta)^\natural$

$\mathcal{G}[[\ G_1 \ \Box\ G_2\ ]] =$
$\lambda e_{\in \mathbf{Env}}.\lambda i_{\in \mathbf{Ind\_Fuz}}.\lambda t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto}\otimes\mathbf{Out}}.$
$(\mathcal{G}[[G_1]]\ e\ i\ t\ a) \uplus (\mathcal{G}[[G_2]]\ e\ i\ t\ a)$

$\mathcal{G}[[\ E \rightarrow S\ ]] =$
$\lambda e_{\in \mathbf{Env}}.\lambda i_{\in \mathbf{Ind\_Fuz}}.\lambda.t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto}\otimes\mathbf{Out}}.$
$(\lambda(t_{1 \in \mathbf{Bool}}, i_{t \in \mathbf{Grd}}).$
$\quad \mathbf{if}\ t_1\ \mathbf{then}\ \mathcal{S}[[S]]\ e\ t(i,i_t)\ t\ a$
$\quad\quad \mathbf{else}\ \{\!\!\{\ a\ \}\!\!\})(\mathcal{B}[[E]]\ e\ t\ a)$

3.4.6 Expressions

In this section we define the classical evaluation functions for the expressions in any imperative language. It is not necessary to know the Ind_fuz. but it is necessary to know the T_op. for the calculation of the degree of the result of the expressions.

$\mathcal{E}$: EXPRESIÓN $\rightarrow$ **Env** $\rightarrow$ **T_op** $\rightarrow$ (**Sto** $\otimes$ **Out**) $\rightarrow$(**EV** $\times$ $((\mathbf{Sto} \otimes \mathbf{Out})_\perp \oplus \delta))^\natural$

$\mathcal{E}[[\ L\ ]] = \lambda e_{\in \mathbf{Env}}.\lambda t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto}\otimes\mathbf{Out}}.$
$\{\!\!\{\ (\mathcal{L}[[\ L\ ]],a)\ \}\!\!\}$

$\mathcal{E}[[\ \Upsilon\ E\ ]] = \lambda e_{\in \mathbf{Env}}.\lambda.t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto}\otimes\mathbf{Out}}.$
$\{\!\!\{\ (\mathcal{U}[[\ \Upsilon\ ]]\ t\ (\mathcal{E}[[\ E\ ]]\ e\ t\ a),a)\ \}\!\!\}$

$\mathcal{E}[[\ E_1\ \Omega\ E_2\ ]] = \lambda e_{\in \mathbf{Env}}.\lambda t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto}\otimes\mathbf{Out}}.$
$\{\!\!\{(\mathcal{B}I[[\ \Omega\ ]]\ t\ (\mathcal{E}[[\ E_1\ ]]\ e\ t\ a)\ (\mathcal{E}[[\ E_2\ ]]\ e\ t\ a),a)\ \}\!\!\}$

$\mathcal{E}[[\ I\ ]] = \lambda e_{\in \mathbf{Env}}.\lambda t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto}\otimes\mathbf{Out}}.$
$\{\!\!\{([(\mathbf{id}_{\mathbf{EV}},a),((\underline{\lambda} l._{\in \mathbf{Loc}}.\mathbf{acc\_alm}\ l\ a),a)$
$,\top,\top,\top]\ (\mathbf{bound}\ I\ e\ a),a)\}\!\!\}$

$\mathcal{E}[[\ I.E\ ]] = \lambda e_{\in \mathbf{Env}}.\lambda t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto}\otimes\mathbf{Out}}.$
$\{\!\!\{\ ([\top,\top,\top,\mathcal{E}[[\ E\ ]]\ e\ t\ a,\top]$
$(\mathbf{bound}\ I\ e\ a),a)\ \}\!\!\}$

$\mathcal{E}[[\ I(E^*)\ ]] = \lambda e_{\in \mathbf{Env}}.\lambda t_{\in \mathbf{T\_op}}.\lambda a_{\in \mathbf{Sto}\otimes\mathbf{Out}}.$
$[\top,\top,[\lambda p_{\in \mathbf{Func}}.(p(\mathcal{E}[[\ E^*\ ]]\ e\ t\ a)\ a),\top],\top,\top] \qquad (\mathbf{bound}[[I]]\ e\ a)$

Monadic operators

$\mathcal{U}$: MONADIC OPERATORS $\rightarrow$ **T_op** $\rightarrow$ **EV** $\circ\!\!\rightarrow$
EV

$\mathcal{U}[[$ **not** $]] = \lambda t_{\in \mathbf{T\_op}}.\lambda(b_{\in \mathbf{Bool}}, g_{\in \mathbf{Grd}})$.
if b **then** (**false**,g′) **else** (**true**,g′)
where g′ is such that t(g,g′) = unit

$\mathcal{U}[[$ **-** $]] = \lambda t_{\in \mathbf{T\_op}}.\lambda(n_{\in \mathbf{Num}}, g_{\in \mathbf{Grd}}).(-n,g)$

Dyadic operators

$\mathcal{BI}$: DYADIC OPERATORS $\rightarrow$ **T_op** $\rightarrow$
(**EV** $\otimes$ **EV**) $\circ\!\!\rightarrow$ **EV**

$\mathcal{BI}[[$ **and** $]] =$
$\lambda t_{\in \mathbf{T\_op}}.\lambda(t_{1 \in \mathbf{Bool}}, g_{1 \in \mathbf{Grd}}, t_{2 \in \mathbf{Bool}}, g_{2 \in \mathbf{Grd}})$.
if t_1 **then** (t_2,t(g_1,g_2))
else (**false**,t(g_1,g_2))

$\mathcal{BI}[[$ **or** $]] =$
$\lambda.t_{\in \mathbf{T\_op}}.\lambda(t_{1 \in \mathbf{Bool}}, g_{1 \in \mathbf{Grd}}, t_{2 \in \mathbf{Bool}}, g_{2 \in \mathbf{Grd}})$.
if t_1 **then** (**true**,t′(g_1,g_2))
else (t_2,t′(g_1,g_2))

$\mathcal{BI}[[$ **op_arit** $]] =$
$\lambda.t_{\in \mathbf{T\_op}}.\lambda(n_{1 \in \mathbf{Num}}, g_{1 \in \mathbf{Grd}}, n_{2 \in \mathbf{Num}}, g_{2 \in \mathbf{Grd}})$.
(**op_arit**(n_1,n_2),t(g_1,g_2))
where **op_arit** $\in$ {**add,minus,times,div**}

$\mathcal{BI}[[$ **op_rel** $]] =$
$\lambda t_{\in \mathbf{T\_op}}.\lambda(n_{1 \in \mathbf{Num}}, g_{1 \in \mathbf{Grd}}, n_{2 \in \mathbf{Num}}, g_{2 \in \mathbf{Grd}})$.
(**op_rel**(n_1,n_2),t(g_1,g_2))
where **op_rel** $\in$ {**<,>,<=,>=,=,!=**}

$\mathcal{BI}[[$ **op_arit**$_r$ $]] =$
$\lambda t_{\in \mathbf{T\_op}}.\lambda(n_{1 \in \mathbf{Real}}, g_{1 \in \mathbf{Grd}}, n_{2 \in \mathbf{Real}}, g_{2 \in \mathbf{Grd}})$.
(**op_arit**$_r$(n_1,n_2),t(g_1,g_2))
where **op_arit**$_r$: **addr, minusr, timesr, divr**

$\mathcal{BI}[[$ **op_rel**$_r$ $]] =$
$\lambda t_{\in \mathbf{T\_op}}.\lambda(r_{1 \in \mathbf{Real}}, g_{1 \in \mathbf{Grd}}, r_{2 \in \mathbf{Real}}, g_{2 \in \mathbf{Grd}})$.
(**op_rel**$_r$(r_1,r_2),t(g_1,g_2))
where **op_rel**$_r$: **<, >, <=, >=, =, !=**

$\mathcal{BI}[[$ **op_arit**$_t$ $]] =$
$\lambda t_{\in \mathbf{T\_op}}.\lambda(n_{1 \in \mathbf{Num}}, g_{1 \in \mathbf{Grd}}, n_{2 \in \mathbf{NumT}}, g_{2 \in \mathbf{Grd}})$.
(**op_arit**$_t$(n_1,n_2),t(g_1,g_2))
where **op_arit**$_r$: **addt, minust, timest, divt**

$\mathcal{BI}[[$ **op_arit**$_{tr}$ $]] =$
$\lambda t_{\in \mathbf{T\_op}}.\lambda(n_{\in \mathbf{Num}}, g_{1 \in \mathbf{Grd}}, r_{\in \mathbf{Real}}, g_{2 \in \mathbf{Grd}})$.

(**op_arit**$_{tr}$(n,r),t(g_1,g_2))
where **op_arit**$_{tr}$:
addtr,minustr,timestr,divtr

$\mathcal{BI}$[[**op_con**]] =
$\lambda t_{\in \mathbf{T\_op}}.\lambda c\_f_{1 \in \mathbf{C\_fuzzy}}.\lambda c\_f_{2 \in \mathbf{C\_fuzzy}}.$
(**op_con**(c_f_1,c_f_2, t))
where **op_con**: **union,intersection**

$\mathcal{BI}$[[**in**]] = $\lambda t_{\in \mathbf{T\_op}}.\lambda n_{\in \mathbf{Num}}.\lambda c\_f_{\in \mathbf{C\_fuzzy}}.$
(**in**(n,c_f))

4 Example

The example that follows aims to show the potential of the language. We have decomposed it into three files in order to show the reusability of the code. The programme has been executed with what we have called strategy 1, i.e. that the initial **Ind_fuz** es 1 and the **T_op** is the function *max*

4.1 Header file 1

The possibility of constructing functions for the labels about and similar is shown in this file. These functions return a trapezoidal number.

```
\       Similar and about.
\       For strategies 1,2,3 use _y
\       for strategies 4,5,6 use _y1
const _y ={0.,0.,0.,0.};
const _y1= {1.,1.,1.,1.};
function similar(INTEGER x1):FUZZY;
const a1=5., b1=10.;
var FUZZY z; REAL co;
begin
    if UNIT = 1.0 then z:=_y
            else z:=_y1
    end if;
    co := EXTR(x1);
    z{1}:=co-a1-b1;
    z{2}:=co-a1;
    z{3}:=co+a1;
    z{4}:=co+a1+b1;
    similar <- z
end;

function about(INTEGER x1): FUZZY
const b1= 4., a1=5.;
var FUZZY z; REAL co;
```

```
begin
    if UNIT = 1.0 then z:=_y
            else z:=_y1
    end if;
    co := EXTR(x1);
    z{1}:=co-a1-b1;
    z{2}:=co-a1;
    z{3}:=co+a1;
    z{4}:=co+a1+b1;
    about <- z
end; end INCLUDE
```

4.2 Header file 2

In this file the capacity of the language to program one the possible algorithms for the comparison of trapezoidal numbers is shown. Only the kernel, and not the supports, are taken into account here.

```
\ Test fuzzy b and r
function cmp_bor(FUZZY b, r): BOOLEAN
    function cmp_rea(REAL re1,re2):BOOLEAN
    begin
        cmp_rea <- re1 < re2
    end;
    function cmp_coin(FUZZY b1,b2):BOOLEAN
    var
        REAL m1,m2,m3;
        BOOLEAN v1,v2;
    begin
        m1 := b1{3} - b2{2};
        m2 := b1{3} - b1{2};
        m3 := m1 / m2 ;
        v2 := true;
        v1 := false;
        if UNIT = 1.
           then DEGREE(v2) := m3;
                DEGREE(v1) := 1. - m3
           else DEGREE(v2) := 1. - m3;
                DEGREE(v1) := m3
        end if;
        v1 := v1; v2:=v2;
        CASE
           true -> cmp_coin <- v1
        [ ] true -> cmp_coin <- v2
        ESAC
    end;
```

```
function cmp_coin1(FUZZY b1,b2):BOOLEAN
var
    REAL m1,m2,m3;
    BOOLEAN v1,v2;
begin
    m1 := b2{3} - b1{2};
    m2 := b1{3} - b1{2};
    m3 := m1 / m2 ;
    v1 := true;
    v2 := false;
    if UNIT = 1.
       then DEGREE(v1) := m3;
            DEGREE(v2):= 1. - m3
       else DEGREE(v1) := 1. - m3;
            DEGREE(v2):= m3
    end if;
    v1 := v1;
    v2:=v2;
    CASE
       true -> cmp_coin1 <- v1
    [ ] true -> cmp_coin1 <- v2
    ESAC
end;
function cmp_conte(FUZZY b1,b2):BOOLEAN
var
    REAL m1,m2,m3;
    BOOLEAN v1,v2;
begin
    m1 := b2{3} - b2{2};
    m2 := b1{3} - b1{2};
    m3 := m1 / m2 ;
    v1 := false;
    v2 := true;
    if UNIT = 1.
       then DEGREE(v1) := m3;
            DEGREE(v2):= 1. - m3
       else DEGREE(v1) := 1. - m3;
            DEGREE(v2):= m3
    end if;
    v1 := v1;
    v2 := v2;
    CASE
        true -> cmp_conte <- v1
    [ ] true -> cmp_conte <- v2
    ESAC
end;
```

```
    var
        BOOLEAN bo1,bo2,bo3,bo4,bo;
begin
    bo1:= cmp_rea(b{3},r{2});  \Test kernel
    bo2:= cmp_rea(r{3},b{2});  \
    bo3:= cmp_rea(b{3},r{3});  \
    bo4:= cmp_rea(b{2},r{2});  \

    if (bo1 or bo2) then
       \ Case |----b----| |--r--| or opposite
        cmp_bor <- false
    else
        if bo3 then
             \ Case    |---b---|
             \       |-----r-----|
            if not(bo4) then
                cmp_bor <- true
            else
             \ Case  |---b---|
             \         |---r---|
                bo := cmp_coin(b,r);
                cmp_bor <- bo
            end if
        else
            if not(bo4) then
                \ Case   |---b---|
                \      |---r---|
                bo := cmp_coin1(b,r);
                cmp_bor <- bo
            else
                \ Case |----b----|
                \        |--r--|
                bo := cmp_conte(b,r);
                cmp_bor <- bo
            end if
        end if
    end if
end; end INCLUDE
```

4.3 Program file

The capacities of the language from the viewpoint of the definition of linguistic variables are shown with this programme. Three linguistic types are declared : Age, Weight and Person.

The type `Age` has an `age` field of type fuzzy and three linguistic variables : `young()`, `middle()` and `old()`.

The type `Weight` has a `weight` field of type fuzzy and three linguistic variables : `thin()`, `average()` and `fat()`.

Each of the above linguistic variables has the same structure : a constant `t` which is used to test an entry and a variable `va` which will the result. We use them to model similar cases to the following fuzzy set:

The type `Person` has two fields : `ed` of type `Age` and `pe` of type `Weight` and a linguistic variable: `suitable()`.

We declare `pe1` of the type `Person`, we assign to it a fuzzy `age` and `weight` and by using the previous functions, we try to classify it as suitable. We try to model the following situation:

```
type Age   = ty_li
                  FUZZY age
              va_li
              young: ()
                  const t={17.,20.,40.,60.};
                  var BOOLEAN va;
              begin
                  va:= cmp_bor(age, t);
                  young <- va
              end |
              middle: ()
                  const t={20.,40.,60.,65.};
```

```
                var BOOLEAN va;
            begin
                va:= cmp_bor(age,t);
                middle <- va
            end |
            old : ()
                const t={40.,60.,85.,85.};
                var BOOLEAN va;
            begin
                va:= cmp_bor(age,t);
                old <- va
            end
        end,
Weight   = ty_li
            FUZZY weight
        va_li
        thin: ()
            const t={40.,45.,70.,90.};
            var BOOLEAN va;
        begin
            va:= cmp_bor(weight,t) ;
            thin <- va
        end |
        middle: ()
            const t={45.,70.,90.,110.};
            var BOOLEAN va;
        begin
            va:= cmp_bor(weight,t);
            middle <- va
        end |
        fat: ()
            const t={70.,90.,130.,130.};
            var BOOLEAN va;
        begin
            va:= cmp_bor(weight,t);
            fat <- va
        end
    end,
Person = ty_li
            Age ed;
            Weight pe
        va_li
        suitable: ()
            var BOOLEAN p_d,p_m,e_j,e_m;
        begin
            e_j := ed.young();
```

```
                e_m := ed.middle();
                p_d := pe.thin();
                p_m := pe.middle();
            CASE
        e_j or p_d -> suitable <- true
    [ ] e_m and p_m -> suitable <- true(0.7)
    [ ] default -> suitable <- false
            ESAC
            end
         end;
var Person pe1 ;
    BOOLEAN bo1,bo2,bo3;
    INTEGER n,m;
begin
    pe1.ed.age:=similar(26);
    pe1.pe.weight:=about(51);
    n:= 58 ;
    m:= 71 ;
    pe1.ed.age:=similar(n);
    pe1.pe.weight:= about(m);
    bo1:=pe1.suitable();
    print("Age-weight = ",pe1.ed.age,
        pe1.pe.weight," Suitable = ",bo1,nl)
end.
```

Producing 9 outputs:

```
<1>:
Age-weight={43,53,63,73}(1){62,66,76,80}(1)
    Suitable FALSE(1)
<2>:
Age-weight={43,53,63,73}(1){62,66,76,80}(1)
    Suitable FALSE(1)
<3>:
Age-weight={43,53,63,73}(1){62,66,76,80}(1)
    Suitable TRUE(1)
<4>:
Age-weight={43,53,63,73}(1){62,66,76,80}(1)
    Suitable TRUE(1)
<5>:
Age-weight={43,53,63,73}(1){62,66,76,80}(1)
    Suitable TRUE(0.6)
<6>:
Age-weight={43,53,63,73}(1){62,66,76,80}(1)
    Suitable FALSE(1)
<7>:
```

```
Age-weight={43,53,63,73}(1){62,66,76,80}(1)
    Suitable TRUE(0.6)
<8>:
Age-weight={43,53,63,73}(1){62,66,76,80}(1)
    Suitable TRUE(1)
<9>:
Age-weight={43,53,63,73}(1){62,66,76,80}(1)
    Suitable TRUE(1)
```

as a consequence of the execution of the programme, whose flow we try to reflect with the following diagram. (We only show those aspects of the programme that can give rise to a multivaluation).

5 Conclusions

In this paper we have formally designed and specified a programming language that takes into account the fuzzy paradigm. The definition is complete and usable in an industrial environment. This language allows for a large number of extensions: pointers, arrays, modules, etc. However, more interesting would be:

To develop in depth the class and objects as support for the linguistic variables.

To introduce some improvement that would prevent an excessive proliferation of stores.

To introduce time, since in general the correspondence between linguistic values and class fuzzy is not static.

To introduce some methods of inference.

To study the parallelization of the language on executing the guarded commands.

References

[1] J.M. Adamo. L.p.l. a fuzzy programming language: 1. syntactic aspects. *Fuzzy Set and Systems*, 3:151–179, 1980.

[2] J.M. Adamo. L.p.l. a fuzzy programming language: 2. semantic aspects. *Fuzzy Set and Systems*, 3:261–289, 1980.

[3] P.D. Moses. Denotational semantics. *In Formals Models and Semantics. (Ed Jan van Leewen). Elsewier. pag.575-629*, 1990.

[4] Daniel Sanchez Alvarez. El lambda cálculo_b. *En Actas del VII Congreso Español sobre Tecnología y Lógica Fuzzy. pag.311-316*, 1997.

[5] Daniel Sanchez Alvarez y Antonio F. Gómez Skarmeta. Semánticas de lenguajes con datos borrosos. *En Actas del VIII Congreso Español sobre Tecnología y Lógica Fuzzy. pag.271-278*, 1998.

[6] R.D. Tennent. *Principles of Programming Languages.* Prentice-Hall, Englewood Cliffs,N.J., 1981.

[7] L.A. Zadeh. The concept of a linguistic variable and its applications to approximate reasoning-I. *Information Sciences*, 8:199–249, 1975.