



Human-centric algebraic machine learning

AML-DL Specification - Revised M18

Deliverable D3.2

WP3 - AML Description Language



This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 952091.

Consortium



ALGEBRAIC AI



Chantalimaud
Foundation



German
Research Center
for Artificial
Intelligence



TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

uc3m



Project

Title: ALMA: Human Centric Algebraic Machine Learning
 Acronym: ALMA
 Coordinator: Proyectos y Sistemas de Mantenimiento SL (eProsima)
 Grant agreement: 952091
 Call: H2020-EIC-FETPROACT-2019
 Program: Horizon 2020
 Start: 01 September 2020
 Duration: 48 months
 Website: <https://alma-ai.eu>
 E-mail: alma@eprosima.com
 Consortium: **Proyectos y Sistemas de Mantenimiento SL (eProsima) (EPROS)**, Spain
Champalimaud Foundation (CF), Portugal
ALGEBRAIC AI (ALG), Spain
German Research Center for Artificial Intelligence (DFKI), Germany
Inria (Inria), France
TU Kaiserslautern (TUK), Germany
FIWARE Foundation e.V. (FF), Germany
Technical Research Centre of Finland (VTT), Finland
Carlos III University (UC3M), Spain

Deliverable

Number: **D3.2**
 Title: **AML-DL Specification - Revised M18**
 Month: 18
 Work Package: WP3 - AML Description Language
 Work Package leader: Algebraic AI (ALG)
 Deliverable leader: Algebraic AI (ALG)
 Deliverable type: Report (R)
 Dissemination level: Public (PU)
 Date of submission: 2022-02-28
 Version: v0.9
 Status: Finished

Version history

Version	Date	Responsible	Author/Reviewer	Comments
v0.1	2021-04	ALG	Fernando Martin-Maroto	Draft
v0.2	2021-04	CF	Gonzalo G. de Polavieja	Draft
v0.3	2021-04	ALG	Nabil Abderrahaman-Elena	Draft
v0.4	2021-05	DFKI	Vitor Fortes Rey	Draft
v0.5	2021-05	ALG	Nabil Abderrahaman-Elena	Review
v0.6	2021-04	CF	Gonzalo G. de Polavieja	Review
v0.7	2021-05	ALG	Fernando Martin-Maroto	Final review
v0.8	2022-01	ALG	Nabil Abderrahaman-Elena	Minor updates
v0.9	2022-02	ALG	Fernando Martin-Maroto	Final review

Executive summary

Algebraic Machine Learning (AML) is a novel machine learning technique based on algebraic representations of data. Unlike statistical learning, AML algorithms are robust regarding the statistical properties of the data and are parameter-free. AML systems can retain what they learn, including sociocultural ethical practices for human–artificial intelligence systems. The aim of the EU-funded ALMA project is to study the mathematical properties of AML and to leverage them to develop a new generation of interactive human-centric machine learning systems. These systems are intended to minimize bias in order to facilitate trust while better enabling distributed collaborative learning.

AML Description Language (AMLDL) v0.1 is the initial specification of the Algebraic Machine Learning Description Language for semantic embeddings into semilattices. A semilattice is an algebra with a single binary operation that is commutative, associative and idempotent. This algebra provides a minimalistic set of syntactic tools: an idempotent operator, an order relation and a set of constants. AMLDL leverages these simple tools to provide an extended set of instructions with two goals: to give the user the capability to describe elaborated embeddings with a concise code, and to establish a methodology to build embeddings for complex problems into semilattices and other algebras with idempotent operators.

Contents

Executive Summary	4
Contents	5
Acronyms	6
1 Introduction	7
2 Overview of AMLDL v0.1	7
2.1 Semilattices	7
2.2 Language	8
2.2.1 AMLDL Grammar	8
2.2.2 AMLDL Commands	9
3 Interpreter implementation	11
4 Command Set	13
4.1 Constant creation	13
4.2 Duple creation	13
4.3 Term formation and manipulation	14
4.4 Auxiliary commands	15
4.5 Additional commands for unmanaged base languages	15
5 Code examples	15
6 The road ahead	16
References	18

Acronyms

AML	Algebraic Machine Learning.
AMLDL	AML Description Language.

1 Introduction

The word *embedding* is often used as a synonym of *encoding*. However, an embedding has a precise mathematical definition; an embedding is a structure-preserving map that allows to identify a mathematical structure within another. An example of structure preserving maps is an injective homomorphism of a group A into another group B. For such identification to be possible A has to be a subgroup of B and only then the homomorphism is an embedding. Mathematicians have extended this notion into something more general. An embedding of A into B can be defined even when A and B are very different mathematical entities with different properties and operations. For example, a group can be embedded within a graph. This concept is known to mathematical logicians as *semantic embedding*[1, 2].

In a semantic embedding there is the notion of *interpretation*[1]. An interpretation is a mathematical Rosetta Stone that explains how to see A inside of B. We could say that while a regular encoding relies entirely in the interpretation, in the case of a semantic embedding that is not necessarily the case. In the example of the subgroup it is clear that A had to be there, inside of B, from the very beginning. In this case the interpretation is only a function that locates A within B.

From regular encodings to homomorphisms there is a range of many possible structure-preserving maps. Suppose we take a novel and encode it into a string of zeros and ones to store it in our computer memory. The interpretation explains how to read letters from the zeros and ones, how to form words with the letters and how to understand the words. Without the interpretation there is little left from the novel into the string of zeros and ones. If instead of a single novel we encode all the volumes of the national library, we may increasingly be able to deduce a larger portion of the meaning of the string of zeros and ones, and with enough corpus we may be able to reconstruct the missing interpretation or some equivalent interpretation. The encoding of the novel has left too much meaning up to the interpretation so a larger additional corpus is necessary to decipher it.

If instead of a novel we encode a computer program into a string of zeros and ones we may be able to deduce an interpretation from a much shorter bit string. This is because the interpretation of the programming language is simpler than the interpretation of the novel. In a semantic embedding we try to find an interpretation that is as simple as possible. Rather than encoding A into B using a fixed interpretation, a semantic embedding consists in finding A into B by discovering an appropriate interpretation.

Searching for an appropriate (simple and accurate) interpretation of A into B is a natural problem that we, humans, are not so familiar with. When we communicate an object A to a listener B using English we start with a language and its interpretation shared between the speaker and B. The communication with B occurs with an interpretation that is mostly fixed. The goal of the Algebraic Machine Learning Description Language (AMLDDL) is to communicate with a very simple listener B, a semilattice. Semilattices live in Plato's world and we do not share with them a language. On the other hand their simple structure allows us to understand them so well that it becomes possible to find simple interpretations for any A.

2 Overview of AMLDL v0.1

2.1 Semilattices

Semilattices have a single operation, the idempotent operation, $\odot$, from which we can derive an order relation $\leq$. We will have to rely only on these two operations to build our embeddings. Although the number of operations is small, we can have access to an unlimited supply of *constants*. The meaning of the order relation and idempotent operator may be different when different constants are involved. The constants thus increase the descriptive power of the idempotent operator.

A constant of an algebra is an element that has a name, so we refer to elements with a name as *constants*. Constants act as primitives for the embedding and their meaning is a key part of the *interpretation*. The

embedding typically starts by selecting a set of constants and choosing one or many meanings for the order relation and the idempotent operator.

To illustrate how constants can modulate the meaning of the order relation suppose that we have constants a and b that refer to two people. We could choose the order relation to mean “is a son of” and write $a < b$ to describe a as a son of b . That is possible but it will leave us without syntactic tools to refer to other family relationships. Instead we can use additional constants *son*, *daughter*, *uncle* and use $a < b \odot \textit{son}$ or $a < b \odot \textit{uncle}$ to modify the meaning of the order relation. Order relationships can also be negative. We can say $a \not< b \odot \textit{daughter}$ to say that a is not the daughter of b .

Every element mentioned in an embedding should be either a constant or an idempotent summation of constants. By putting constants together with the idempotent operator every element can be formed. Elements can be seen as sets of constants. However, two (or more) different sets of constants may yield the same element or may be related by the order relation even if one is not a subset of the other. We refer to pairs of elements related by the order relation as *positive duples* and to the pairs of elements related by the negation of the order relation as *negative duples*.

2.2 Language

AMLDL commands declare constants, construct elements putting constants together, or specify order relationships between pairs of elements. In addition to these basic operations, AMLDL 0.1 has a flexible system of indexing and iteration that allows for a single statement to describe entire families of duples of the same sign.

2.2.1 AMLDL Grammar

```

<const_decl> ::= 'C(' <ID> ')'
<const_vec_decl> ::= 'CV(' <ID> ',' <INT> ')'
<vec_decl> ::= 'V(' <ID> ')'
               | 'V()'
<it_decl> ::= 'T(' <descr> ')'
               | 'T(' <descr> ',' <INT> ')'
<cmp_create> ::= 'CMP(' <comp_create_arg> ',' <comp_create_arg> ')'
<comp_create_arg> ::= <const_decl>
                    | <const_vec_decl>
                    | 'F(' <ID> ')'
                    | 'F(' <ID> ',' <INT> ')'
<cmp_get> ::= 'CMP(' <cmp_get_arg> ')'
<cmp_get_arg> ::= <const_decl>
                 | <const_vec_decl>
                 | <it_decl>
                 | 'F(' <ID> ')'
                 | 'F(' <ID> ',' <INT> ')'
<descr> ::= 'F(' <ID> ')'
            | 'F(' <ID> ',' <INT> ')'
            | <it_decl>
            | <m_expr>
            | <const_decl>

```

```

| <const_vec_decl>
| <vec_decl>
| <cmp_get>

<incl> ::= 'INCL(' <descr> ',' <descr> ')'
<excl> ::= 'EXCL(' <descr> ',' <descr> ')'
<m_expr> ::= 'M(' <m_params> ')'
<m_params> ::= <m_params> ',' <descr>
| <descr>

<iterable> ::= <const_vec_decl>
| <vec_decl>
| <it_decl>
| 'F(' <ID> ')'

<some_cmd> ::= 'SOME(' <iterable> ',' <FLOAT> ',' <BIN> ',' <BIN> ')'
<region_cmd> ::= 'REGION(' <INT> ')'
<header_cmd> ::= 'HEADER(' <ID> ')'
<add_v> ::= 'APP(' <descr> ',' <descr> ')'
<remove_v> ::= 'R(' <descr> ',' <descr> ')'
| 'R(' <descr> ',' <INT> ')'
<save_cmd> ::= '^' <descr>
<free_cmd> ::= '/' <descr>

<statement> ::= <const_decl>
| <const_vec_decl>
| <vec_decl>
| <cmp_create>
| <incl>
| <excl>
| <some_cmd>
| <region_cmd>
| <header_cmd>
| <add_v>
| <remove_v>
| <save_cmd>
| <free_cmd>

<statement_seq> ::= <statement_seq> <statement>
| ε

<program> ::= <statement_seq>

```

2.2.2 AMLDL Commands

AMLDL 0.1 data types are constants, vectors of constants, elements, vectors of elements, duples, and vectors of duples. Vectors may or may not be indexed. Indexed vectors result from the application of the iterator command T .

Commands $C(name)$ and $CV(name, n)$ define a constant and a vector of constants of length n , respectively. To retrieve these elements by name, we use the find operator $F(name)$. Vectors of elements can be defined as $v = V([name])$ and can be built by appending elements using $APP(v, e)$ that appends e to vector v .

To calculate the idempotent operation we can use the command $M(p1, p2, \dots)$ where $p1, p2, \dots$ are constants.

Indexing and iterating along a vector can be done with the help of the iterator command $T(v)$. For example, assume v is a vector with elements $v[0], v[1], \dots, v[n-1]$. Then $M(T(v), e)$ produces a vector $v[0] \odot e, v[1] \odot e, \dots, v[n-1] \odot e$. Iterators can have one or many indexes associated to them. An index is identified by a positive whole number. For example, if v and w are vectors of the same size, $M(T(v, 1), T(w, 1), e)$ returns vector $v[0] \odot w[0] \odot e, v[1] \odot w[1] \odot e, \dots, v[n-1] \odot w[n-1] \odot e$, a vector of size n . We now assume that w has size m . If we use different indexes, $M(T(v, 1), T(w, 2), e)$ returns $v[0] \odot w[0] \odot e, v[0] \odot w[1] \odot e, \dots, v[0] \odot w[m-1] \odot e, \dots, v[1] \odot w[0] \odot e, \dots, v[n-1] \odot w[m-1] \odot e$, a vector of size $n \times m$.

Positive and negative duples are defined using $INC(a, b)$ and $EXC(a, b)$ respectively, which directly translates into $a \leq b$ and $a \not\leq b$. The iterator command associates indexes that remain after a command is applied. For example $INC(a, M(T(v), e))$ produces the vector of n duples: $a \leq v[0] \odot e, a \leq v[1] \odot e, \dots, a \leq v[n-1] \odot e$. If we invoke $INC(T(w, 1), M(T(v, 1), e))$ assuming vector v and w have the same size then we obtain the vector of n duples: $w[0] \leq v[0] \odot e, w[1] \leq v[1] \odot e, \dots, w[n-1] \leq v[n-1] \odot e$. Accordingly, using different indexes, $INC(T(w, 2), M(T(v, 1), e))$ will produce $n \times m$ positive duples: $w[0] \leq v[0] \odot e, w[0] \leq v[1] \odot e, \dots, w[0] \leq v[n-1] \odot e, w[1] \leq v[0] \odot e, \dots, w[m-1] \leq v[n-1] \odot e$.

A few auxiliary commands are available; $R(v, d)$ that removes component d from a copy of vector v , $SOME(v, p)$ that produces a vector with components chosen at random from vector v with probability p , and $CMP(p1, p2)$ that declares a bidirectional map between objects $p1$ and $p2$ such that $CMP(p1)$ returns $p2$ and $CMP(p2)$ returns $p1$. CMP acts componentwise if $p1$ and $p2$ are vectors of the same size.

3 Interpreter implementation

A viable implementation consists of a few commands implemented as functions of a base language like Python or C++. Using a base language has obvious advantages as the entire base language is available to add control loops, variables etc. Relying on a base language makes the implementation simpler as it makes unnecessary to use a parser or a lexer.

A single descriptor object can be used for the commands' input and output. AMLDL commands correspond to functions of the base language that take one or many descriptor objects and return a descriptor object. It is enough with a few types of descriptor objects: type `CONST`, type `SET`, type `VECTOR`, type `CVECTOR`, type `TVECTOR`, and type `DUPLE`.

Descriptors of type `CONST` correspond to constants defined using the command `C` while descriptors of type `CVECTOR` correspond to vectors of constants returned by the command `CV`.

The return values of commands `C`, `CV`, `INC`, `EXC` are *final* descriptors, in the sense that they are appended to the program output. The program output is comprised of a set of descriptors of type `CONST` and a set of descriptors of type `DUPLE`. Other commands, like `M`, `S`, `T`, `V` or `R` return descriptors that are not appended to the program output and that are meant to be used as inputs of other commands or assigned to a variable. In other words, these commands return descriptors that should be *bound*. Hence, descriptors are either final or bound. The binding of a descriptor to an input parameter of a command may occur directly or by means of a variable of the base language.

At any point, the program state corresponds to a set of final descriptors collected so far, and a dictionary that maps names to descriptor objects. Names are regular character string that can be used to invoke descriptor objects by name. The find command `F` is provided to specifically retrieve a descriptor by name although, for convenience, names can always be used instead of descriptor object as parameters of any command.

The command `V([name])`, that declares a vector, produces a descriptor that can be optionally bound. As a side effect, an only if a name is provided, a newly created empty vector is associated with said name. It returns a descriptor of type `VECTOR`.

Descriptors of type `SET` are returned by the commands `M` and `S`. A `SET` descriptor corresponds to a set of constants. When a `VECTOR` or `CVECTOR` is provided as an input to `M` or `S` their components are added to the returned set. The `M` command, which correspond to the idempotent operator, adds every provided parameter to the returned set. These sets are going to be interpreted as the component constants of terms, so they will become elements of the semilattice.

Descriptors of type `DUPLE` correspond to duples and can be positive when produced by the `INC` command, or negative when produced by the `EXC` command. Type `DUPLE` descriptors are always final. The `INC` and `EXC` commands require inputs of type `SET`.

The iterator command `T` provides indexing capabilities and makes the language significantly more flexible. It can operate with any other command. The iterator command changes descriptors of type `SET`, `VECTOR` or `CVECTOR` into descriptors of type `TVECTOR`. When a command has an input of type `TVECTOR`, the command is applied not to the `TVECTOR` itself but to each one of its components. The function then returns a `TVECTOR` whose components are the corresponding returned values.

A command may receive one or many inputs of type `TVECTOR`. Descriptors of type `TVECTOR` have an index number. If a command receives more than one input of type `TVECTOR` with the same index the components are replaced at the same time in all of them. In this case, all the `TVECTOR` descriptors with the same index should have the same dimension otherwise an error is returned. For example, a command that receives one or many descriptors of type `TVECTOR` of size n and the same index, returns a `TVECTOR` of size n . When a command receives more than one input of type `TVECTOR` with different

indexes the output corresponds to a TVECTOR with components of type TVECTOR producing a tree of depth equal to the number of different indexes used. For example, if a command receives two descriptors of type TVECTOR with sizes m and n then it produces a TVECTOR of size m whose components are descriptors of type TVECTOR of size n , therefore producing a total of $m \times n$ applications of the command.

Descriptors form a tree. We have just mentioned the case of nested descriptors of type TVECTOR. Descriptors of type DUPL consist of a pair of references to descriptors of type SET, one to the left and the other to the right hand descriptors. Descriptors of type VECTOR have components of any type except DUPL.

If a managed language like Python is used as base language we do not need to worry about releasing the memory of the bound descriptors once they are used. However, if an unmanaged language like C or C++ is our base language we have to be more careful. Leaving the responsibility of memory management to the user will make the writing of clean composite expressions such as $M(R("a"), T("c"))$ impossible. If, on the other hand, all the bound descriptors are stored for automatic release after interpreting the full AMLDL code the memory usage tends to grow too large. A possible solution (simpler than writing a garbage collector) is to establish that a bound descriptor is deleted each time it is used. In order to reuse the same descriptor (which is rarely required) additional commands may be necessary: one command to prevent deletion, command $\wedge p$, and another to release the memory once the descriptor is no longer needed, the command $/p$. AMLDL bindings for unmanaged languages require such commands.

4 Command Set

4.1 Constant creation

C(name)	Defines a constant with the given <i>name</i> . Returns a descriptor of type CONST that cannot be bound.
CV(name, n)	Defines a vector of constants with names: name[0], name[1], ..., name[n - 1]. Returns a descriptor of type CVECTOR of length <i>n</i> that cannot be bound.
V([name])	Defines an empty vector. Optional parameter <i>name</i> assigns a name to the vector. Returns a descriptor of type VECTOR that can be optionally bound.
F(name[,i])	Finds a descriptor with the given <i>name</i> . Optional parameter <i>i</i> is an integer. <i>name</i> should refer to a descriptor of type CONST, VECTOR or CVECTOR. Returns the descriptor associated to the name. Returns the <i>i</i> -th component of the descriptor associated to the <i>name</i> , if <i>i</i> is given and the descriptor is of type VECTOR or CVECTOR.

4.2 Duple creation

INC(l, h)	Defines a positive duple $l \leq h$. <i>l</i> and <i>h</i> are descriptors or descriptor names of any type except type DUPL. Returns a final descriptor of type DUPL or TVECTOR that cannot be bound.
EXC(l, h)	Defines a negative duple $l \not\leq h$. <i>l</i> and <i>h</i> are descriptors or descriptor names of any type except type DUPL. Returns a final descriptor of type DUPL or TVECTOR that cannot be bound.

4.3 Term formation and manipulation

The arguments p_i in the following commands are descriptors or the names assigned to descriptors.

M(p[,...])	Calculates an idempotent summation. The argument or arguments p are descriptors of any type except of type DU- PLE. Returns a descriptor of type SET. If some argument is of type TVECTOR, M returns a descriptor of type TVEC- TOR whose components are of type SET. The return value should be bound.
APP(p, pa)	Appends pa to p. Argument p should be of type VECTOR. Modifies p. No return value.
R(p, pr)	Removes pr from a copy of the vector p. If argument pr is an integer, removes the pr-th component. Argument p is not modified. Argument p is of type VECTOR or TVECTOR. Returns a descriptor of the same type than p that should be bound.
CMP(p1,p2)	Declares $p1$ and $p2$ as the logical negation of each other. $p1$, $p2$ are of type CONST or CVECTOR of the same length. Operates component-wise if $p1$, $p2$ are CVECTORS. No return value.
CMP(p)	Finds or calculates a descriptor that is the complementary of p. p is a descriptor of type CONST, CVECTOR or TVECTOR. Copies p recursively replacing every CONST descriptor by its complementary. The argument p is not modified. Returns a descriptor of the same type as p.
T(p[, i])	Associates an iterator to p. The argument p is of any type except DU- PLE. The optional parameter i is an integer that identifies the iterator. Returns a descriptor of type TVECTOR that should be bound.

4.4 Auxiliary commands

HEADER(name)	Guards a section of the program that should be read just once. Returns <i>True</i> when <i>HEADER(name)</i> has not been read yet.
SOME(p, prob [, atLeastOne, notAll])	Randomly chooses items with a probability <i>prob</i> . Default values for optional parameters are <i>False</i> . Boolean parameter <i>atLeastOne</i> ensures at least one item is selected. Boolean parameter <i>notAll</i> makes sure not all the items are selected. The argument <i>p</i> is a descriptor, or the name assigned to a descriptor, of type VECTOR, CVECTOR or TVECTOR. Returns a descriptor of the same type than <i>p</i> .
REGION(n)	Assigns duples defined after the statement to region <i>n</i> , where <i>n</i> is a positive integer. The default region is 0. The assigned value remains unaltered until REGION is set to a different value.

4.5 Additional commands for unmanaged base languages

$\hat{p}$	Prevents the deletion of a descriptor <i>p</i> .
/ <i>p</i>	Returns the memory of a preserved descriptor $\hat{p}$.

5 Code examples

In this first example, two vectors of *n* constants, *black* and *white*, are defined and the independence of each constant with the rest is provided as a set of negative constraints. Iterator command *T* is used twice with the same index *i* resulting in a coordinated replacement. The following code returns the two CVECTORS of *n* constants

```
1 CV("black", n)
2 CV("white", n)
```

For instance, if *n* = 3, the code defines the constants:

$$black[0], black[1], black[2], white[0], white[1], white[2].$$

These CVECTORS are used below to define two sets of negative duples with:

```
1 EXC(T("black", i), M("white", R("black", T("black", i))))
2 EXC(T("white", i), M("black", R("white", T("white", i))))
```

which result in the following duples:

$$black[0] \not\leq white[0] \odot white[1] \odot white[2] \odot black[1] \odot black[2]$$

$$black[1] \not\leq white[0] \odot white[1] \odot white[2] \odot black[0] \odot black[2]$$

$$black[2] \not\leq white[0] \odot white[1] \odot white[2] \odot black[0] \odot black[1]$$

$$white[0] \not\leq black[0] \odot black[1] \odot black[2] \odot white[1] \odot white[2]$$

$$white[1] \not\leq black[0] \odot black[1] \odot black[2] \odot white[0] \odot white[2]$$

$$white[2] \not\leq black[0] \odot black[1] \odot black[2] \odot white[0] \odot white[1].$$

The next example makes a random selection of constants from *black* and calculates its idempotent summation. For each constant not selected in *black*, the corresponding constant in *white* is appended to the summation so exactly n constants are always added. The resulting element for the summation is assigned to the right hand side of a positive duple. First we define the constants, and set the *black* and *white* CVECTORS as being complementary component by component,

```
1 n = 3
2 C("r")
3 CV("black", n)
4 CV("white", n)
5 COMP("black", "white")
```

then we carry out a random selection of components of *black* that get stored in a descriptor x of type VECTOR, and use x to define a positive duple with the code:

```
1 p = 0.05
2 atLeastOne = False
3 notAll = True
4 x = SOME("black", p, atLeastOne, notAll)
5 INC("r", M(x, R("white", COMP(x))))
```

A possible return for x could have components $black[0], black[1]$ and the positive duple would then be:

$$r \leq black[0] \odot black[1] \odot white[2].$$

6 The road ahead

The goals of AMLDL are to give the user the capability to describe elaborated algebraic semantic embeddings with a concise program, and to establish a methodology to build embeddings for real life problems. We have focused in version v0.1 on the first goal. We have started with an small set of commands that allow expressing large embeddings with a short AMLDL code. By using nested iterator commands, a single AMLDL statement can translate into a large set of ordered duples. This has proven quite useful in practice. In addition, a few other auxiliary commands, COMP, R, REGION or SOME, have been included in 0.1 for convenience as the need for them became apparent while writing embeddings for AML test problems. AMLDL 0.1 suffices to write any embedding for finite models.

Future specifications of AMLDL should pay more attention to the second goal. Finding algebraic semantic embeddings can be regarded as a skill that the user should learn. Although AMLDL may never be able to fully remove the need for a user with experience, there are, hopefully, primitives and methodologies that can be integrated in AMLDL to make the embedding process more intuitive. The embedding of a problem often requires the dissection of the problem into smaller parts. Some parts are common to many problems and these common parts are motifs for which specific AMLDL syntax could be provided.

For future specifications to fulfill the second goal, we first need to develop the formal basis of what constitutes a good embedding. We are at present working on this problem and the the first paper on the topic of semantic embeddings into semilattices is planed for early 2022[3]. This paper will give the mathematical principles of what constitute a good embedding, as well as a classification of embeddings into four different categories: *complete*, *tight*, *explicit* and *concise*. For example, a method to transform any embedding into a tight and explicit embedding is detailed, so future versions of AMLDL could include commands to transform embeddings automatically. Explicit embeddings have a much higher chance to produce useful models.

Future specifications should address the task of facilitating the use of natural and rational numbers. With AMLDL v0.1 is possible to embed natural and rational quantities but with the cost of using many

constants. There is a need for higher level language primitives to facilitate the use of numerical variables. Current specification is mostly useful for embeddings of Boolean type variables.

There is also a need for logic quantifiers. In the coming paper the embedding of first order universal quantifiers through the combined use of *explicit* embeddings and *context constants*[3] is proposed. Existential quantifiers can potentially be embedded by using skolemization[4, 2], a well-known technique in mathematical logic. The inclusion of powerful higher level syntax in AMLDL for positive existential quantifiers seems quite feasible while negative existential quantifiers may remain out of reach for much longer. Future versions of AMLDL would ideally include a syntax as close as possible to first order logic quantifiers.

Finally, there is a need for infinite models. Infinite models are necessary for scientific and engineering problems that require the practical and conceptual handling of unbounded quantities. Assuming an infinite set of constants is not enough to address the problem and the need for other algebras beyond semilattices seems inescapable. This implies that the task of calculating models of other algebras with idempotent operators should be undertaken in the near future. For example, we know that semilattices extended with a finite set of unary operators suffice to embed models of the infinite chain of natural or rational numbers. Most interesting is the fact that the AML techniques, including full and sparse crossing[5, 6] or pinning[5, 6] seem to apply to these extended algebras in a quite straightforward manner.

Algebraic semantic embeddings is a fascinating topic mostly unexplored. Historically, semantic embeddings have been used as tools to make proofs for decidability problems meant to study the limits of formal languages. We believe that algebraic semantic embeddings have an enormous potential for machine intelligence and for natural sciences in general. We hope that AML and AMLDL will contribute to popularize the algebraic semantic embedding technique and to make it practical to resolve real life problems.

References

- [1] Stanley Burris and H. P. Sankappanavar. *A course in universal algebra*. Clarendon Press, 1981.
- [2] David Marker. *Model Theory: An Introduction*. Springer-Verlag, 2000.
- [3] Fernando Martin-Maroto and Gonzalo G. de Polavieja. “Semantic Embeddings in Semilattices”. In: *In preparation* (2022).
- [4] Wilfred Hodges. *A shorter model theory*. Cambridge University Press, 1997.
- [5] Fernando Martin-Maroto and Gonzalo G. de Polavieja. “Algebraic Machine Learning”. In: *arXiv:1803.05252* (2018).
- [6] Fernando Martin-Maroto and Gonzalo G. de Polavieja. “Finite Atomized Semilattices”. In: *arXiv:2102.08050* (2021).