The Markov Algorithmic Machine

Markov algorithms (MA for short, also called Normal Algorithms) stand as a model of associative computation based on pattern matching and substitution. The model is equivalent to other models of computation, such as Turing Machines and Lambda Calculus that constitute mathematical foundations of various classes of programming languages. The class of languages circumscribed by the MAs addresses mainly rule-based languages (such as CLIPS) useful for knowledge oriented applications. However, these languages can be seen as general purpose, offering a mix of declarative and imperative programming flavour. They are equipped with interesting off-track data representation and control features that allow for direct coding of high abstraction solving strategies. The solution of a problem is much on the side of the problem description (i.e. it is declarative) rather than being distorted by the question of how to use the control constructs of the programming language for solving the problem.

Structure

The building blocks of a Markov Algorithmic Machine (**MAM** for short) are:

- the data register (**DR**), containing a string **R** of symbols,
- the control unit (**CU**), and
- the algorithm store (**AS**) that stores the Markov algorithm (**MA**).

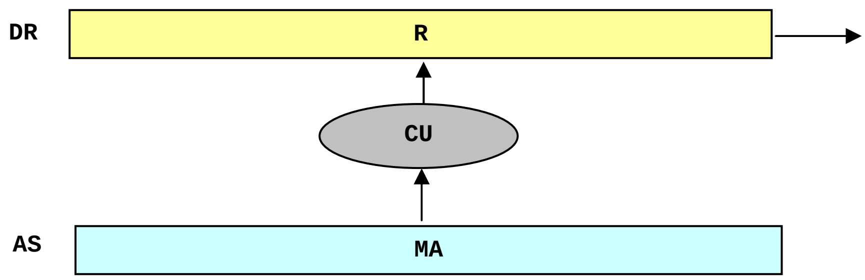

Figure 1. The block structure of MAM

Data

The **MAM** works with strings of symbols. The data register **DR** stores a string, called **R**, from the set $\{A_b \cup A_l\}^*$, where

- A_b is the base alphabet;
- A_l is the local (working) alphabet;
- $A_b \cap A_l = \emptyset$.

The sets A_b and A_l cannot contain reserved symbols that are used to encode **MAM** algorithms. The data register has an unlimited capacity and extends to the right as much as necessary.

The initial string in **DR** (before the algorithm stored in **AS** starts) and the final string in **DR** (after the al the algorithm terminates) must be in A_b^*. The string from DR can contain symbols from A_l during the execution of the algorithm only.

Rules

The basic building block of a Markov algorithm is the "associative substitution rule" of the form:

```
rule ::= identification_pattern -> substitution_pattern [.]
                   LHS                        RHS
identification_pattern ::= symbol*
substitution_pattern ::= symbol*
symbol ::= constant | generic_variable | local_variable
```

A ***constant*** is a symbol from A_b

A ***local_variable*** is a symbol from A_l

A ***generic_variable*** is a conventional symbol that at – during the execution of the Markov algorithm – stands for a symbol from a subset of A_b. By convention, generic variables are noted by the letter **g**, possibly decorated by subscript and/or superscript indices. The set of all legitimate values a generic variable **g** can be bound to is called the domain of the variable and is noted **Dom(g)**. The following restrictions apply:

- During the execution of an MA, a generic variable from a rule can be bound to a unique symbol from its domain while the rule is applied.
- The scope of a generic variable spans the algorithm within which it appears.
- Any generic variable from its RHS must also occur in the LHS of a rule.

Note that a rule can be textually terminated by a dot. Such a rule is a terminal rule. If applied (see the comments below on how the control unit works) it stops the MAM.

Algorithms

A Markov algorithm is mainly an ordered set of rules, known as the body of the algorithm, enhanced with declarations that:

- structure A_b into subsets and
- specify the domains of the generic variables used in the body of the algorithm.

By convention an algorithm is described as follows:

```
algorithm ::= name base_alphabet_declaration;
       [generic_var_declaration;]*
       [label: rule;]*
end [name]

base_alphabet_declaration ::= ([set [, set]*])
set ::= subset_of_Ab | (set) | set set_constructor set
subset_of_Ab ::= subset_name | {constant [,constant]*}
set_constructor ::= ∪ | ∩ | \
generic_var_declaration ::= set generic_var [, generic_var ]*
label ::= natural_number
```

Rules are numbered according to their position in the algorithm. We assume that the first rule has the label **1** whereas the **i**-th rule has the label **i**.

By convention, a symbol that occurs in a rule and that is not declared as a constant from **A_b** is considered a local variable.

The syntax of an **MA** is of little importance as far as its textual description makes it clear which are the domains of generic variables and which is the role of the symbols used in the rules of the algorithm (constants from **A_b**, local variables, generic variables). As an example, the algorithm **set_difference** removes from the string **R** (stored in **DR**) all symbols that are in the set **B**. When the algorithm terminates the **R** contains symbols from **A\B** only.

```
set_difference(A,B); B g1;
    1: g1->;
    2: ->.;
end
```

The Control Unit (CU)

The behaviour of the control unit relies on two concepts: rule applicability and rule application (or rule firing).

Definition 1. (rule applicability)
Let **r:** $a_1\ a_2\ \ldots\ a_n \text{ -> } b_1\ b_2\ \ldots\ b_m$ be a rule of a Markov algorithm with the alphabet $A_b \cup A_l$ and the generic variables **G**. The rule **r** is applicable if and only if there is a substring $c_1\ c_2\ \ldots\ c_n$ in **DR** such that for each $i \in 1..n$ precisely one of the following conditions holds:

1. $a_i \in A_b \wedge a_i = c_i$;
2. $a_i \in A_l \wedge a_i = c_i$;
3. $a_i \in G \bullet (\forall j \in 1..n \mid a_j = a_i \bullet c_j \in Dom(a_i) \wedge c_j = c_i)$, i.e. the variable a_i is bound to a unique value from its domain.

Definition 2. (rule application)
Let **r:** $a_1\ a_2 \ldots a_n \text{ -> } b_1\ b_2 \ldots b_m$ be a rule of a Markov algorithm with the alphabet $A_b \cup A_l$ and the generic variables **G**. Let **s:** $c_1\ c_2 \ldots c_n$ be a substring in **DR** which makes the rule applicable. The application of **r** on **s** is the substitution of **s** by a substring $q_1\ q_2 \ldots q_m$ computed from the string $b_1\ b_2\ \ldots\ b_m$ in the following way:

1. $q_i = b_i$, if $b_i \in A_b$;
2. $q_i = b_i$, if $b_i \in A_l$;
3. $q_i = c_j$, if $b_i \in G \wedge b_i = a_j$.

Example. Let $A_b = \{1,2,3\}$, $A_l = \{x,y\}$, $Dom(g_1)=\{2\}$, $Dom(g_2)= A_b$ and consider that the string in the data register **DR** is **R= 1111112x2y31111**. The rule **r:** $1g_1xg_1yg_2 \text{ -> } 1g_2x$ is applicable. The string that is matched by the identification pattern of the rule is **12x2y3** and the values bound to the generic variables are $g_1 \leftarrow 2,\ g_2 \leftarrow 3$. Before the application (the rule **r** is applicable but is not yet applied) the matching of rule **r** against the string **R** is as shown below.

```
R: 11111 1 2  x 2  y 3 1111
       r: 1 g1 x g1 y g2 -> 1g2x
```

After the application of the rule (the rule **r** is effectively applied) the string **R** is:

```
R: 1111113x1111
```

Note that there is a major difference between the notions of applicability and application. A rule can be made applicable by more than one substring from **DR**. Indeed, **DR** can contain several substrings that match the identification pattern of the rule. However, the rule is applied for only one substring that made it applicable.

The following convention eliminates the ambiguity of which matched substring fires the rule: if there are several strings that trigger the rule (made it applicable) then the rule is fired (applied) for the leftmost triggering string from **DR**.

The **CU** of a **MAM** is wired for a very simple control strategy of executing an **MA**. In the control algorithm below, **Rules** is an ordered set that designates the rules from the body of the executed Markov algorithm, and **R** is the string from **DR**.

```
control(R,Rules) {
     i:= 1; n := card(Rules);
     CU_status := running;
     while i ≤ n and CU_status = running
     {
             r := the i-th rule from Rules;
             if r is applicable then
             {
                    R:= fire the rule r;
                    // application of r has side effects on R

                    if r is a terminal rule
                    then CU_status := terminate
                    else i:= 1
             }
             else i:=i+1
     }
     if CU_status = terminate
     then return R
     else error: the algorithm is blocked
}
```

The control algorithm above shows that the rules of an algorithm are NOT executed sequentially. They are only tested sequentially for applicability. If a non-terminal rule is applied, **MAM** resumes the testing of rule applicability from the top of algorithm's body (rule **#1**). The execution strategy is similar to that of repeatedly pushing the contents of **DR** through a layered sieve as in figure 2.

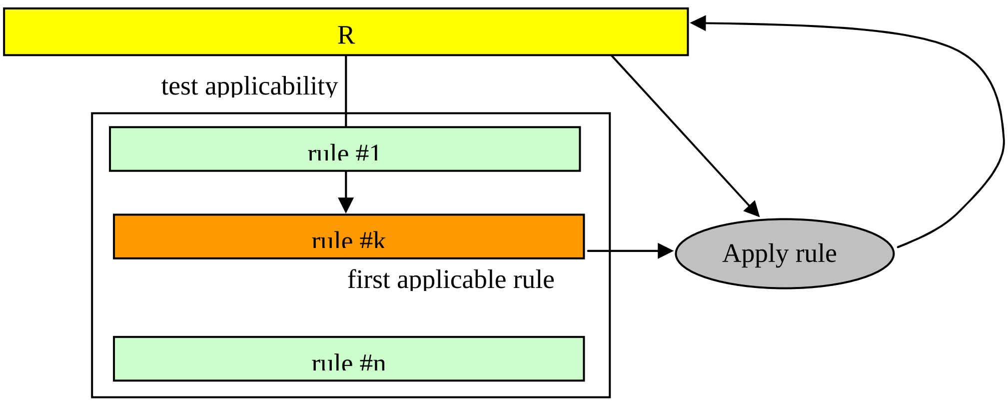

Figure 2. The CU control strategy

Layer **#k** of the sieve corresponds to rule **#k** from the algorithm. If the contents of **DR** triggers the layer **#k**, then the rule **#k** is applied, **DR** is updated by substitution, and then the contents of the resulting **DR** is pushed once more into the device at the top of the layered sieve.

A Markov algorithm does not rely on any of the conventional control mechanisms (sequencing, conditional execution, looping) provided by the conventional programming languages. The only actions that are performed are pattern matching (parameterised string identification, from left to right in **DR**) and textual substitution. The control is *data-driven*. However, **MAM** is equivalent as far as the computation power is concerned to a Turing machine. Said in other words, a problem solved using a Turing machine can also be solved by **MAM** and vice-versa. Since the class of number theoretic functions that are Turing computable is the class of recursive functions, it follows that **MAM** can solve any problem the mathematical model of which is a recursive function.

An example

As a simple Markov algorithm consider reversing a string made of symbols from a set **A**. The algorithm uses as local variables two symbols **a,b** $\notin$ **A**.

```
reverse(A); A g1,g2;
    1: ag1g2 -> g2ag1;
    2: ag1 -> bg1;
    3: abg1 -> g1a;
    4: a ->.;
    5: -> a;
end reverse
```

Before the start of **reverse**, consider that the content of DR is the string **NOW**. The execution of the algorithm follows the steps below. Each step corresponds to the application of a rule and is represented as:

string **R** before rule application **– rule label ->** string **R** after the rule application

```
R:   NOW -5-> aNOW -1-> OaNW -1-> OWaN -2-> OWbN
         -5-> aOWbN -1-> WaObN -2-> WbObN
         -5-> aWbObN -2-> bWbObN
         -5-> abWbObN -3-> WabObN -3-> WOabN -3-> WONa -4->
     WON
```