Control Flow in the Markov Algorithmic Machine

The practice with conventional algorithms biases our understanding of control towards the traditional control constructs such as: sequencing (a sort of functional composition), selection, and looping. As a matter of fact, these control constructs are rooted in the von Neumann computer architecture. Quite differently, Markov algorithms have a very simple predefined control flow. It is therefore interesting to answer a basic question: do the conventional control structures have a counterpart in the realm of the Markov Algorithmic Machine? Otherwise said what could be solved by a conventional computer can be solved as well with **MAM**?

The answer is affirmative. Here we sketch the way two essential programming mechanisms can be modeled in **MAM**: functional composition and addressable memory. Selection and looping can be modeled in similar ways.

First we define some basic modifying mechanisms of **MA**s. To begin with, note that an **MA** with the alphabet $A_b \cup A_l$ can be considered as a string made with symbols from an extended base alphabet: $A_b \cup A_l \cup$ **Reserved_symbols**. Therefore, an **MA** can be processed by a meta-**MA** and so on. In this way, the **MAM** is an unbounded layered machine. The meta-operations explained below can be carried out by a meta-**MA** on a string which corresponds to an **MA**.

Algorithm concatenation

Let M_1 and M_2 be two Markov algorithms such that:

- Both algorithms have the same alphabet $A_b \cup A_l$ (A_l can be seen as the union of the two sets of local variables of M_1 and M_2);
- The common generic variables have the same domain;

We also expect that the processing of M_1 is consistent with the processing of M_2, i.e. the algorithms carry out parts of a consistent transformation that could be done by a larger algorithm. Consider that M_1 has N_1 rules and M_2 has N_2 rules. We note $M_1;M_2$ the concatenation of the two algorithms such that the resulting algorithm, say **M**, has:

- The alphabet $A_b \cup A_l$;
- The generic variables of both algorithms;
- N_1+N_2 rules: $r_{11}\ r_{12}\ ...r_{1N1}\ r_{21}\ r_{22}...r_{2N2}$, where the rules of M_2 are re-labeled to follow the rule labels of M_1.

For example, consider the following algorithms:

```
M1(A,B); B g1;        M2(A,B);
   1: g1 ->;            1: ->.
end                   end
```

The algorithm M_1 is blocked on termination, while M_2 performs nothing. However the algorithm $M = M_1;M_2$ makes sense and computes a string made up with symbols from the set **A\B**.

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

Alphabet images

Consider an algorithm **M** which works with the alphabet $A = A_b \cup A_l$. Let $A^0, A^1, \ldots, A^n, \ldots$ be an infinite sequence of alphabets such that:

1. $A^i \cap A^j = \varnothing$ for any $i \neq j$.
2. There is a one to one function $f: A^i \to A^j$, for any $i \neq j$.
3. $A^0 = A$

The alphabet A^i is the i^{th} image of the alphabet **A**. **A** is the image **0** of itself. Similarly, A_b^i and A_l^i are the i^{th} images of A_b and respectively of A_l. The images of the alphabet of an algorithm help the implementation of explicit control mechanisms.

Conventions.

1. If **g** is a generic variable defined over a subset **D** of values from A_b (ie. **D** is the domain of **g**), then the corresponding generic variable defined over $D^i \subseteq A_b^i$ (i.e. the image **i** of the base alphabet A_b) is noted g^i.

2. If an **MA** contains a declaration $\texttt{Domain}^i\ g_1^i, g_2^i, \ldots$. then we assume that the algorithm implicitly contains the declarations $\texttt{Domain}^j\ g_1^j, g_2^j, \ldots$ for all $j \geq i$.

3. If a variable g^i with the domain D^i from a rule is bound to a symbol $s^i \in D^i$ then the varible g^j from the rule, if present, is automatically bound to the symbol $s^j \in D^j$. In addition, if the variable g^j occurs in the ***identification_pattern*** of the rule, s^j must be the symbol from **DR** register identified by the variable.

For example, in the algorithm below it is not compulsory to declare that the generic variables g_1^1 and g_2^1 are defined over A_b^1. The declaration $A_b^1\ g_1^1, g_2^1$ is implied by the declaration $A_b\ g_1, g_2$ as are implied all the declarations $A_b^i\ g_1^i, g_2^i$ with $i > 0$.

```
M(A_b∪{a}); A_b g_1,g_2;
    1: ag_1 -> ag_1^1;
    2: g_1^1g_2 -> g_1^1g_2^1;
    3:  ->.;
end
```

Algorithm transformation

Algorithm transformation stands for the process of modifying one or several from the following elements of an **MA**:

- The alphabet;
- The domains of the generic variables;
- The rules of the algorithm.

Algorithm transformation can be performed by a meta-algorithm, as mentioned. However, here we are interested only in specifying the transformation rather than describing the meta-algorithm that is going to perform the transformation. The following notation is adopted to specify the transformation of an algorithm **M**:

$$M_{transformed} = M\frac{(\textit{initial\_alphabet} \ / \ \textit{modification\_patterns})}{(\textit{modified\_alphabet} \ / \ \textit{modified\_patterns})}$$

where ***modification_patterns*** is a list of (non parameterized) strings that can appear in **M**. These strings are textually substituted by the corresponding ***modified_patterns***. For simplicity, if new symbols appear in the ***modified_alphabet*** we consider that these symbols extend the base alphabet of $M_{transformed}$. For example, consider the algorithm **M** with $A_b = A \cup B$ (where **A** and **B** may not be disjoint) and $A_l=\{a\}$. The alphabet of **M** is $A_b \cup A_l = A \cup B \cup \{a\}$.

```
M(A,B); A g1; B g2;
   1: ag1 -> g1a;
   2: ag2 -> a;
   3: a ->.;
   4:  -> a;
end
```

Assume that we want to transform **M** such that it works with the first image of the alphabet $A \cup B \cup \{a\}$ and, when stopping, plants a new symbol, say **b**, $b \notin A^i \cup B^i \cup \{a^i\}$, $i \geq 0$, in the **DR** register. Therefore, we ask that the alphabet of **M** is changed to its first image and then extended with the symbol **b**, and that the third rule is modified to **3:** a^1 **-> b.;** Call **M'** the transformed algorithm. The specification of the required modification is:

$$M' = M\frac{(A \cup B \cup \{a\} \ / \ .)}{((A \cup B \cup \{a\})^1 \cup \{b\} \ / \ b.)}$$

By convention, modifying the alphabet of **M** from $A \cup B \cup \{a\}$ to $(A \cup B \cup \{a\})^1$ means implicitly modifying the domains of the generic variables, the local variables, and the rules of the algorithm so that the resulting algorithm is able to work with the new alphabet. The modified algorithm is:

```
M'(A^1,B^1,{b}); A^1 g1^1; B^1 g2^1;
   1: a^1g1^1 -> g1^1a^1 ;
   2: a^1g2^1 -> a^1;
   3: a^1 -> b.;
   4:  -> a^1;
end
```

Notice that care must be paid to distinguish between the reserved symbols of the algorithm **M** and the reserved symbols used in the specification. For example, the round parentheses above are considered different from round parentheses of an **MA**. For convenience, here both groups of parentheses are represented identically although we should have used different symbols.

The notation used for specifying algorithm transformation is restrictive. As a matter of fact the format of the general specification amounts to some kind of meta-rules and, therefore, parameterized ***modification_patterns*** and ***modified_patterns*** would seem appropriate. However, the restricted form of specification suits the current goal of the discussion and keeps the presentation simple.

Functional composition

Consider two terminating algorithms M_1 and M_2 with the same alphabet $A= A_b \cup A_l$. We have to construct an algorithm $M = M_2 \circ M_1$ such that $M(R)= M_2(M_1(R))$. Clearly the meta-operation $\circ$ implements the sequencing $M_1\ M_2$.

To start the construction of M, consider that there are two symbols, **a** and **b**, which do not occur in any image of **A**. In addition, let **R** be the initial string in **DR**.

Step 1.
Build the algorithm M_1' that behaves as M_1 except for the fact that it does not terminate. Instead, on termination of M_1, M_1' inserts the constant **a** somewhere in the register **DR** of **MAM** and then gets blocked.

$$M_1' = M_1 \frac{(A\ /\ .)}{(A \cup \{a,b\}\ /\ a)}$$

Step 2.
The string in **DR** is $M_1(R) \in A_b^*$, including the constant **a** that is placed somewhere in **DR**. It is now possible to start the processing of M_2. What we have to do is to apply the algorithm M_2 on the string $M_1(R)$. However, there is a problem: how to assemble M_1' and M_2?

Variant 1. If we concatenate $M_1';M_2$ then it won't work. After M_1' completes, there might be a string produced by M_2 that triggers again some rules from M_1'. In particular, the rules with an empty identification pattern would be fired endlessly.

Variant 2. If we concatenate $M_2;\ M_1'$ then it works as far as M_2 is blocked during the processing done by M_1'. The blocking of M_2 is simple. We transform M_2 into an algorithm M_2' that is able to work with the alphabet A^1 only. There must be an additional algorithm, call it H_0, which - when M_1' terminates (i.e. when M_1' generates the constant **a** in **DR**) - converts the alphabet of the string in **DR** from A_b to A_b^1.

```
$H_0(A_b \cup \{a,b\})\ A_b\ g_1,g_2;$
   1: $g_1 a \to a g_1$;       // move a to the left margin of DR
   2: $a g_1 \to a g_1^1$;      // convert the right neighbor of a
   3: $g_1^1 g_2 \to g_1^1 g_2^1$; // propagate conversion to the right in DR
end
```

Now we can build the algorithm $M_{12}= H_0;M2';M1'$, where

$$M_2' = M_2 \frac{(A\ /\ :\to)}{(A^1 \cup \{a,b\}\ /\ :a \to a)}$$

Observe how M_{12} works. Initially, there is no symbol **a** in **DR** and the string in **DR** is made from symbols of the image **0** of A_b. Therefore, H_0 and M_2' are blocked: H_0 cannot start working in the absence of symbol **a**, and M_2' cannot work with the image **0** of **A** and in absence of symbol **a**. The algorithm M_1' starts working and terminates by inserting the symbol **a** somewhere in **DR**.

The algorithm H_0 is now able to start working and changes the alphabet of the string in **DR** from A_b to A_b^1. The symbol **a** is not altered, except from being moved to the left margin of **DR**.

Once H_0 is blocked, M_2' starts working and does the job of M_2. Notice that M_2' must work in the presence of the symbol **a** positioned to the left margin of **DR**. However, the rules of the original algorithm M_2 were not written for such a case. A rule with the format ***label*:-> *substitution_ pattern*** (i.e. a rule that has an empty identification pattern) will produce wrong results when applied. Indeed, the symbol **a** will be dragged between the strings produced by M_2 and hence by M_2'. In order to avoid this to happen, all the rules of the form **:->*substitution_pattern*** must be changed to **:a->a *substitution_pattern***. Moreover, this transformation prevents the rules with an empty **LHS** from M_2' from firing before M_1' terminates.

When M_2' terminates, the string in **DR** is made with symbols from A_b^1 instead of symbols from A_b and, in addition, it contains the symbol **a** positioned at the left margin of **DR**. Therefore,

$$M_{12}(R) = a(M_2(M_1(R)))^1$$

Step 3.
Transform the alphabet of the string in **DR** from A_b^1 to A_b and then eliminate the symbol **a** from **DR**. Assume that this transformation is made by an algorithm called H_1. For H_1 to be able to start, M_{12} must not terminate. Since H_1 will be concatenated at the top of M_{12} additional protection must be taken to block H_1 until M_{12} finishes its job: H_1 is able to start working only when the special symbol **b** occurs somewhere in **DR**.

```
H1(Ab∪{a,b}) Ab g1,g2;
   1: g1^1 b -> bg1^1;   // move b to the left margin of DR
   2: bg1^1 -> bg1;      // convert the right neighbor of b
   3: g1g2^1 -> g1g2;    // propagate conversion to the right in DR
   4: ab -> .;           // terminate processing and get rid of a,b
end
```

We can build the algorithm:

$$M = H_1;\ H_0;\ M_2 \frac{(A\ /\ :\text{->},\ .\ )}{(A^1 \cup \{a,b\}\ /\ :a\text{->}a,\ b)}\ ;\ M_1'$$

Replacing M_1' by its formula we obtain the final form of the functional composition.

$$M_2 \circ M_1 = H_1;\ H_0;\ M_2 \frac{(A\ /\ :\text{->},\ .\ )}{(A^1 \cup \{a,b\}\ /\ :a\text{->}a,\ b)}\ ;\ M_1 \frac{(A\ /\ .)}{(A \cup \{a,b\}\ /\ a.)}$$

Indeed, $(M_2 \circ M_1)(R) = M_2(M_1(R))$.

An example

Design an **MA** that performs the following operation on natural numbers: **2(n + 1)**

1. We agree to represent a natural number as a string of symbols, e.g. the number **3** is represented by **xxx**, where **x** stands for a symbol different from the reserved symbols of an **MA**. Zero is represented by the empty string etc.

2. The algorithm for computing the successor of a natural number is trivial:

```
succ({x});
   1:->x.;
end
```

3. The algorithm for doubling a number **n** is also simple and uses the local variable **&**:

```
double({x});
   1: &x -> xx&;
   2: &->.;
   3: ->&;
end
```

4. The wanted algorithm is **M = double o succ**.

Addressable memory

The basic **MAM** has no directly addressable memory. Addressing is purely associative. However, on the basis of functional composition, it is quite easy to implement a sort of direct addressing mechanism.

First we build a processing focusing mechanism that makes possible the application an algorithm on specific substrings from **DR**.

Let **M** be a terminating **MA** with the alphabet $A=A_b \cup A_l$ and **a** and **b** two symbols not present in any image of **A**. Note **a]M[b** the algorithm with the following behavior:

$$a]M[b\ (S_l a S b S_r) = S_l a M(S) b S_r,$$

where **S**, S_l and S_r are strings made with symbols from **A** and not containing the special symbols **a** and **b**. It is easy to see that **a]M[b** restricts the action of **M** to the string enclosed between the symbols **a** and **b**. Therefore, the symbols **a** and **b** act as an address for the string **S** which has to be processed by **M**.

The transformation of **M** to **a]M[b** uses functional composition. Considering that the composition operator **o** is right associative we have:

$$a]M[b = L_1 \text{ o } M \frac{(A\ /\ :\text{->})}{(A^1 \cup \{a,b\}/\ a:\text{->}a)} \text{ o } L_0$$

The auxiliary algorithms L_0 and L_1 are:

$\mathtt{L_0(A_b \cup \{a,b\});\ A_b\ g_1,g_2;}$

  $\mathtt{1:\ ag_1\ \text{->}\ ag_1^{1};}$

  $\mathtt{2:\ g_1^{1}g_2\text{->}\ g_1^{1}g_2^{1};}$

  $\mathtt{3:\ b\text{->}b.;}$

end

$\mathtt{L_1(A \cup \{a,b\});\ A_b\ g_1,g_2;}$

  $\mathtt{1:\ ag_1^{1}\ \text{->}\ ag_1;}$

  $\mathtt{2:\ g_1g_2^{1}\text{->}\ g_1g_2;}$

  $\mathtt{3:\ b\text{->}b.;}$

end

The direct addressing memory can now be implemented by choosing an appropriate set of symbols **`#1,#2,....#n`** acting as addresses in **`DR`**. If we want to apply an algorithm **`M`** on the content of the location **`#k`**, we have to write **`#k]M[#k+1`**.

Labeled Markov Algorithms

The focusing mechanism of a Markov algorithm is restricted to the data register **`DR`** only. Explicit addressing would be of interest within the body of the algorithm itself. Addressing the rules explicitly is a clear step towards making an **`MA`** look similar to a conventional program that uses conditional and unconditional jumps. What follows shows that an algorithmic machine able to process such algorithms - called labeled algorithms, or **`LMA`** for short - can be built starting from basic Markov algorithms. In addition, any **`LMA`** has an equivalent **`MA`**.

Assume that in an **`LMA`** the rules have the following extended syntax:

rule ::= <u>identification_pattern</u> -><u> substitution_pattern</u> **{[.] | ,*label*}**

The ***label*** following the ***<u>substitution_pattern</u>*** in a rule **`r`** shows what rule is to be tried next after the application of **`r`**. Therefore, the control algorithm of a labeled **`MAM`** is slightly modified as follows:

```
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:= label of r
                     // the next rule to be tried, explicitly
                     // specified by the label of rule r
           }
           else i:=i+1
       }

       if CU_status = terminate
       then return R
       else error: the algorithm is blocked
}
```

When a rule is not applicable the algorithm continues in sequence. After the application of a rule the control does not go to the top of the algorithm, as in the case of **`MAM`**. Instead, there is an unconditional jump to another rule that is explicitly specified. The control of a **LMA** is made explicit. For example, consider the **LMA** variant of the **reverse** algorithm.

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

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

Theorem. The class of **LMA**s is equivalent to the class of **MA**s. In other words for any **LMA** there is an equivalent **MA** and vice-versa.

Proof.

1. The first part of the proof is to show that any **LMA** can be converted into a basic **MA**. In other words, the explicit control can be fully converted to the data-driven control.

Consider a (terminating) **LMA M** with **N** rules and with the alphabet $A = A_b \cup A_l$. In addition, assume there is a set of symbols $J = \{a_i \mid i=1,N+1\} \cup \{b_i \mid i = 1,N\}$ such that $A^i \cap J = \varnothing$, for any $i\geq 0$. For each rule **i** of **M** we build an equivalent basic Markov algorithm, where the symbols from **J** are local variables and the base alphabet is **A**.

Non-terminal rule **i:*string* -> *string'*,j**	Terminal rule **i:*string* -> *string'*.**
M_i(A) declarations of M; A g; 1: g a_i -> a_i g; 2: a_i -> b_i; 3: b_i *string* -> a_j *string'*; 4: b_i g -> g b_i; 5: b_i -> a_{i+1}; End	M_i(A) declarations of M; A g; 1: g a_i -> a_i g; 2: a_i -> b_i; 3: b_i *string* -> *string'*.; 4: b_i g -> g b_i; 5: b_i -> a_{i+1}; end

In addition, we build an algorithm AM_0 whose role is to unblock the application of the algorithm AM_1, which corresponds to the first rule of **M**.

```
AM0(A) declarations of M; A g;
    1: -> a1;
end
```

It is easy to see that the basic Markov algorithm $M' = M_1;M_2;...;M_n;M_0$ is - behaviorally - equivalent to the labeled Markov algorithm **M**. Therefore, the labeled Markov algorithm is reducible to a basic **MA**.

2. Any **MA** has an equivalent **LMA**. This is obvious since each terminal rule of the given **MA** stays the same in the equivalent **LMA**, and any non-terminal rule

```
i: identification_pattern -> substitution_pattern;
```

of the **MA** can be rewritten

```
i: identification_pattern -> substitution_pattern,1;
```

Indeed, if the rule **i** is applied, the control engine of **MAM** tries again all the rules of the given **MA** from the first rule to the next applicable one. ■

Concluding remarks

We implemented functional composition and, therefore, sequencing on the basis of the simple associative processing mechanism present in **MAM**. We also implemented a sort of direct addressing scheme in **DR** and within the body of an **MA**. In similar ways, selection and controlled looping can also be implemented. Therefore, any conventional algorithm that has an explicit control flow, relying on sequencing, selection and looping, can be built as a Markov algorithm. The computational power of **MAM** is equivalent to that of a conventional computer.

However, apart from this kind of informal justification of the computational power of **MAM**, the theorem above makes it possible to formally prove that the class of functions computable with **MAM** corresponds precisely to the class of partial recursive functions.

There are programming languages based on the **MAM** model. The processing mechanisms of such languages go beyond the conventional habits of named data structures and the conventional control constructs (explicit control statements). Essentially, the processing relies on substitution driven by pattern matching of data. The problem universe is represented using structures of symbols. Parts of the universe, identified associatively, are then replaced by new symbolic structures that can inherit from the originals. This kind of processing leads toward a declarative programming paradigm. A rule *identification_ pattern -> substitution_ pattern* is seen as a relationship between generic parts of the problem universe. Although this type of processing can be analyzed as well from the viewpoint of logic programming, it is treated here as a standalone associative programming paradigm.