

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 and necessary to answer a basic question: do the conventional control structures have a counterpart in the realm of MAM? 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 need some more basic modifying mechanisms of MAs. To begin with, note that an MA with the alphabet $A_B \cup A_1$ can be considered as a string made with symbols from an extended base alphabet: $A_B \cup A_1 \cup \text{Reserved_symbols}$. Therefore, an MA can be processed by a meta-MA and so on. In this way, the MAM is an infinitely 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 $A_B \cup A_1$ (A_1 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 $A_B \cup A_1$;
- The generic variables of both algorithms;
- n_1+n_2 rules: $r_{11} r_{12} \dots r_{1n_1} r_{21} r_{22} \dots r_{2n_2}$, 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;
1: g1 ->;
end

M2(A,B);
1: ->.
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 $A \setminus B$.

```

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

```

Alphabet images

Consider an algorithm \mathfrak{M} which works with the alphabet $\mathfrak{A} = \mathfrak{A}_b \cup \mathfrak{A}_1$. Let $\mathfrak{A}^0, \mathfrak{A}^1, \dots, \mathfrak{A}^n, \dots$ be an infinite sequence of alphabets such that:

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

The alphabet \mathfrak{A}^i is the i^{th} image of the alphabet \mathfrak{A} . \mathfrak{A} is the image 0 of itself. Similarly, \mathfrak{A}_b^i and \mathfrak{A}_1^i are the i^{th} images of \mathfrak{A}_b and respectively of \mathfrak{A}_1 . 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 \mathfrak{A}_b , then the corresponding generic variable defined over $D^i \subseteq \mathfrak{A}_b^i$ (i.e. the image i of the base alphabet \mathfrak{A}_b) is noted g^i .
2. If an MA contains a declaration $\text{Domain}^i \ g_1^i, g_2^i, \dots$ then we assume that the algorithm implicitly contains the declarations $\text{Domain}^j \ g_1^j, g_2^j, \dots$ 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 variable 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 D^j 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 \mathfrak{A}_b^1 . The declaration $\mathfrak{A}_b^1 \ g_1^1, g_2^1$; is implied by the declaration $\mathfrak{A}_b \ g_1, g_2$; as are implied all the declarations $\mathfrak{A}_b^i \ g_1^i, g_2^i$; with $i > 0$.

```

M( $\mathfrak{A}_b \cup \{a\}$ );  $\mathfrak{A}_b \ g_1, g_2$ ;
1:  $ag_1 \rightarrow ag_1^1$ ;
2:  $g_1^1 g_2 \rightarrow g_1^1 g_2^1$ ;
3:  $\rightarrow .$ ;
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 :

$$M_{\text{transformed}} = M \frac{(initial_alphabet / modification_patterns)}{(modified_alphabet / modified_patterns)}$$

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 $M_{\text{transformed}}$.

For example, consider the algorithm m with $A_b = A \cup B$ (where A and B may not be disjoint) and $A_1 = \{a\}$. The alphabet of m is $A_b \cup A_1 = A \cup B \cup \{a\}$.

```
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: $a^1 \rightarrow b.$; Call m' the transformed algorithm. The specification of the required modification is:

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

Modifying the alphabet of m from $A \cup B \cup \{a\}$ to $(A \cup B \cup \{a\})^1$ means 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'(A1,B1,{b}); A1 g11; B1 g21;
1: a1g11 -> g11a1;
2: a1g21 -> a1;
3: a1 -> b.;
4: -> a1;
end
```

Notice that care must be paid to distinguish between the reserved symbols of m (an MA) 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 $A = A_b \cup A_1$. 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 one or several 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 .

```
H0(A_b ∪ {a,b}) A_b g1,g2;
  1: g1a -> ag1;          // move a to the left margin of DR
  2: ag1 -> ag1^1;       // convert the right neighbor of a
  3: g1^1g2 -> g1^1g2^1; // propagate conversion to the right in DR
end
```

Now we can build the algorithm $m_{12} = H_0; m_2'; m_1'$, where

$$m_2' = m_2 \frac{(A / :->)}{(A^1 \cup \{a, b\} / :a->a)}$$

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 \circ 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 o 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. Any rule with the format *label*:-> *substitution_pattern* (i.e. any rule that has an empty identification pattern) will produce wrong results. 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} an 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(A_b ∪ {a,b}) A_b 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:

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

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

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

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 $M = \text{double} \circ \text{succ}$.

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 $A = A_b \cup A_1$ and a and b two symbols not present in any image of A . Note $a]m[b$ the algorithm with the following behavior:

$$a]m[b (s_1 a s b s_r) = s_1 a m(s) b s_r,$$

where s , s_1 and s_r are strings made with symbols from A_b 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 \circ is right associative we have:

$$a]m[b = L_1 \circ M \xrightarrow{(A / :->)} \circ L_0$$

$$(A^1 \cup \{a, b\} / a :-> a)$$

The auxiliary algorithms L_0 and L_1 are:

<pre> L₀(A_b ∪ {a,b}); A_b g₁,g₂; 1: ag₁¹ -> ag₁¹; 2: g₁¹g₂¹-> g₁¹g₂¹; 3: b->b.; end </pre>	<pre> L₁(A ∪ {a,b}); A_b g₁,g₂; 1: ag₁¹ -> ag₁; 2: g₁g₂¹-> g₁g₂; 3: b->b.; end </pre>
---	---

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 MA has an equivalent LMA.

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

rule ::= identification pattern -> substitution pattern {[.] | ,label}

The *label* following the substitution pattern 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 if 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 LMAs is equivalent to the class of MAs. 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 \mathcal{M} with N rules and with the alphabet $\mathcal{A} = \mathcal{A}_b \cup \mathcal{A}_1$. In addition, assume there is a set of symbols $\mathcal{J} = \{a_i \mid i=1,N+1\} \cup \{b_i \mid i = 1,N\}$ such that $\mathcal{A}^i \cap \mathcal{J} = \emptyset$, for any $i \geq 0$. For each rule i of \mathcal{M} we build an equivalent basic Markov algorithm, where the symbols from \mathcal{J} are local variables and the base alphabet is \mathcal{A} .

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

In addition, we build an algorithm \mathcal{AM}_0 whose role is to unblock the application of the algorithm \mathcal{AM}_1 , which corresponds to the first rule of \mathcal{M} .

```
 $\mathcal{AM}_0(\mathcal{A})$  declarations of  $\mathcal{M}; \mathcal{A} g;$ 
  1: -> a1;
end
```

It is easy to see that the basic Markov algorithm $\mathcal{M}' = \mathcal{M}_1; \mathcal{M}_2; \dots; \mathcal{M}_n; \mathcal{M}_0$ is - behaviorally - equivalent to the labeled Markov algorithm \mathcal{M} . Therefore, the labeled MAM is reducible to the basic MAM.

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 correspond 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 associative processing is specific to the class of logic programming systems, it is treated here as a standalone programming paradigm.