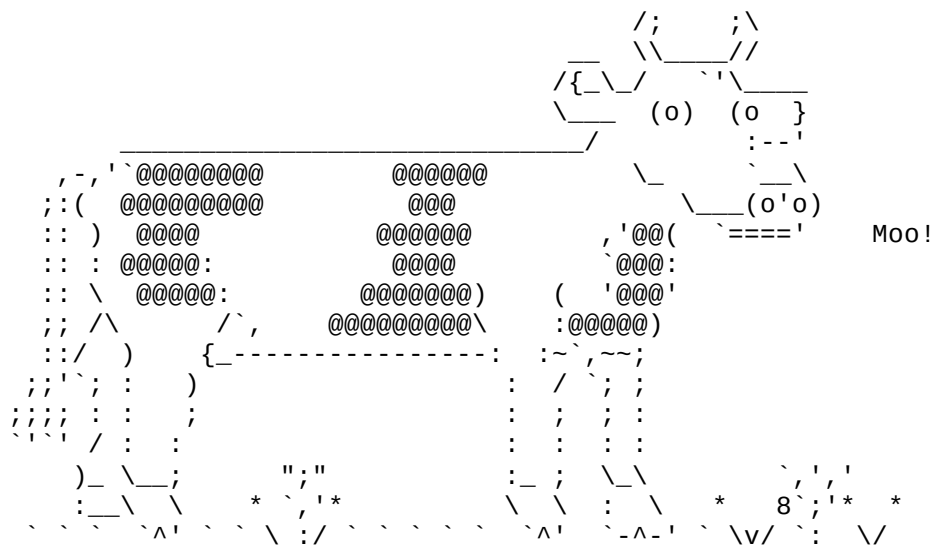
Semantics of COW



July 2004

Alex van Oostenrijk and Martijn van Beek

Table of Contents

Table of Contents 2
Summary 3
Language Specification 3
Syntax 3
State 4
  Accessing the state 4
Natural Semantics 5
  Trivial instructions 5
  Loop-instructies 6
    Loop execution 6
    Forward search 6
    Example of a nested loop 7
  Instruction generation 7
Structural Operational Semantics 9
  Trivial instructions 9
  Loop instructions 9
    Loop execution 9
    Forward search 9
  Instruction generation 10
Comparison of semantics denotations 10
Turing Completeness 11
  Computable functions 11
  Proof structure 11
  Function mechanism 11
  Basic functions 13
  Functional composition 13
  Primitive recursion 13
  Minimalisation 14
  Result 15
References 16

Summary

In this paper we describe the semantics of the programming language “COW.” COW is a fun language developed within the *Esoteric Programming Languages Ring*, in which all keywords are some form of *MOO*. We present natural and structural operational semantics for the language. We also show that COW is Turing complete.

Language Specification

The following table contains all instructions available in COW, according to [Heber]. It is important to note that COW syntax does not enforce any particular instruction order. Instructions can be written in arbitrary order and still make a correct COW program. This has nasty consequences for loops, because loops consist of two instructions (`MOO` and `moo`) that belong together. In the remainder of this text we discuss the problems that arise in the semantics and how they can be solved.

Code	Instruction	Meaning
0	`moo`	This command is connected to the MOO command. When encountered during normal execution, it searches the program code in reverse looking for a matching MOO command and begins executing again starting from the found MOO command. When searching, it skips the instruction that is immediately before it (see MOO).
1	`mOo`	Moves current memory position back one block.
2	`moO`	Moves current memory position forward one block.
3	`mOO`	Execute value in current memory block as if it were an instruction. The command executed is based on the instruction code value (for example, if the current memory block contains a 2, then the moO command is executed). An invalid command exits the running program. Value 3 is invalid as it would cause an infinite loop.
4	`Moo`	If current memory block has a 0 in it, read a single ASCII character from STDIN and store it in the current memory block. If the current memory block is not 0, then print the ASCII character that corresponds to the value in the current memory block to STDOUT.
5	`MOo`	Decrement current memory block value by 1.
6	`MoO`	Increment current memory block value by 1.
7	`MOO`	If current memory block value is 0, skip next command and resume execution after the next matching moo command. If current memory block value is not 0, then continue with next command.
8	`OOO`	Set current memory block value to 0.
9	`MMM`	If no current value in register, copy current memory block value. If there is a value in the register, then paste that value into the current memory block and clear the register.
10	`OOM`	Print value of current memory block to STDOUT as an integer.
11	`oom`	Read an integer from STDIN and put it into the current memory block.

Syntax

The grammar of COW is specified as follows:

S ::= moo | mOo | moO | mOO | Moo | MOo | MoO | MOO | OOO | MMM | OOM | oom | S_1 S_2.

Here, S is short for statement. It is not necessary to separate statements from each other in a composition, since all COW statements have a length of precisely three characters so that a parser is able to determine where a statement begins and where it ends.

Since `MOO` and `moo` form a while-loop together, it seems logical to join them as one syntactical construction. Unfortunately, we cannot do so because the instruction `mOO` can produce individual `MOO`s and `moo`s, so that (half) while-loops can be formed dynamically. It is also possible for a program to end while still inside a while-loop!

State

We represent the *state* as a 6-tuple s = (M, p, r, V, W, l) where:

- M is an infinite list m_0, m_1, … with $m_i \in \mathbb{Z}$, which represents the memory (data storage). At the start of program execution, this list is filled with zeros.
- $p \in \mathbb{N}$ is the memory pointer. It points to the active memory element $m_p \in M$. The memory points is initially 0 and can never be negative..
- $r \in \mathbb{Z} \cup \varepsilon$ is the *register* (see the `MMM` instruction). The register is initially empty (ε).
- V is a finite list of input values $v_0,\ldots,v_n$ where $v_i \in \mathbb{Z}$, representing the input values that are given to the program using the keyboard (from STDIN). At the start of program execution the list contains zero or more elements ($n \geq 0$).
- W is a finite list of output values w_0, …, w_n where $w_i \in \mathbb{Z}$, representing the output values of the program (sent to STDOUT). The list W is initially empty (n = 0).
- l is the *nesting level*, necessary for searching through while-loops (see natural semantics). At the start of program execution, the nesting level is 0.

We choose to model input and output as part of the state, so that the semantics of a program with a given input (as a list of numbers) can be determined. Moreover, the input from `stdin` and output to `stdout` play an important role in COW. We want to be able to study the execution of a program by evaluating the final state of a program; this final state will contain the input and output lists after execution of the program has completed.

Accessing the state

Since the state is complex and not simply a function Var $\rightarrow \mathbb{Z}$, like in the language While (see [Nielson92]), we must find a new method to access and modify the state. This method may consist of a number of functions with signature State $\rightarrow$ State (actually comparable to the substitution operation used in [Nielson92]).

In order to execute instructions in COW, we need access to all elements in the state, and not just the memory M. This means that we need a function for each type of modification. As an example, the function decrements the memory pointer by 1:

$$\delta(m,p,r,v,w,l) = \begin{cases} (m,p-1,r,v,w,l) & \text{if } p > 0 \\ (m,p,r,v,w,l) & \text{otherwise} \end{cases}$$

Of course, we can think of more convenient denotations:

$$\delta(s) = \begin{cases} s[p \rightarrow p\ -1] & \text{if } p > 0 \\ s & \text{otherwise} \end{cases}$$

This is possible because the functions generally work on only one part of the state. The definitions that increment, decrement or set the current memory block's value to zero, and the functions that add numbers to the output list or read numbers from the input list are trivial to construct.

We must be careful that no work is done in the function definitions that should be done in the specification of the natural semantics. We could have a function that evaluates the instruction `Moo`:

4	`Moo`	If current memory block has a 0 in it, read a single ASCII character from STDIN and store it in the current memory block. If the current memory block is not 0, then print the ASCII character that corresponds to the value in the current memory block to STDOUT.

This is possible because all the required information is stored in the state, but the effect of `Moo` can also be expressed in the rules for natural semantics, which is cleaner and more elegant. For this reason, we choose to define only very simple functions to modify the state. These functions will have no names. We will use the following intuitive syntax instead:

$$<\text{Moo}, S> \rightarrow s[m_p \rightarrow v_0, v \rightarrow \text{tail}(v)] \quad \text{als } m_p = 0$$

One can imagine an s prefix, as in $s.m_p$, so that the denotation resembles records in the Clean programming language. For operations on lists (arrays), we use the following simple operators taken from Clean:

Function	**Meaning**	**Example**
++	List concatenation	[1,2] ++[3,4] → [1,2,3,4]
head	First element of a list	head [1,2,3] → 1
tail	List minus its first element	tail [1,2,3] → [2,3]

Natural Semantics

The natural semantics of COW is easy to write, were it not for the instruction `mOO` (3) which wrecks the day. Since we were unable to enforce complete while-loops in the syntax (but allow two halves instead), we have to deal with this problem in the natural semantics.

Trivial instructions

All instructions save 0, 3 and 7 are trivial to describe, and so is the composition of statements:

$$[\text{comp}] \quad \frac{<S1, s> \rightarrow s', \quad <S2, s'> \rightarrow s''}{<S_1\ S_2, s> \rightarrow s''}$$

1	$[mOo^{p>0}]$	$<mOo, s>$	$\rightarrow s[p \rightarrow p - 1]$	if $p > 0$
1	$[mOo^{p=0}]$	$<mOo, s>$	$\rightarrow s$	if $p = 0$
2	[moO]	$<moO, s>$	$\rightarrow s[p \rightarrow p + 1]$	
4	$[Moo^{in}]$	$<Moo, s>$	$\rightarrow s[m_p \rightarrow v_0, v \rightarrow \text{tail}(v)]$	if $m_p = 0$
4	$[Moo^{out}]$	$<Moo, s>$	$\rightarrow s[v \rightarrow v + m_p]$	if $m_p \neq 0$
5	[MOo]	$<MOo, s>$	$\rightarrow s[m_p \rightarrow m_p - 1]$	
6	[MoO]	$<MoO, s>$	$\rightarrow s[m_p \rightarrow m_p + 1]$	
8	[OOO]	$<OOO, s>$	$\rightarrow s[m_p \rightarrow 0]$	
9	[MMM]	$<MMM, s>$	$\rightarrow s[r \rightarrow m_p]$	if $r = \varepsilon$
9	[MMM]	$<MMM, s>$	$\rightarrow s[m_p \rightarrow r, r \rightarrow \varepsilon]$	if $r \neq \varepsilon$
10	[OOM]	$<OOM, s>$	$\rightarrow s[w \rightarrow w ++ m_p]$	
11	[oom]	$<oom, s>$	$\rightarrow s[m_p \rightarrow v_0, v \rightarrow \text{tail}(v)]$	

Loop-instructies

We have to do more work for instructions 0 and 7 (`moo` and `MOO`), because of the existence of half while-loops. We will solve this using an environment, in which we store the body of a while-loop. In fact, we store all the statements that follow `MOO` (including `MOO` itself). The environment env is a list of statements (initially empty). When a while-loop begins, the instruction `MOO` places all the statements that follow it in the enviroment. The end of the loop, moo, retrieves all the statements from the environment and executes them, thereby execution the loop again. Since loops may be nested, it is necessary that the environment contains a list of statements. An example of an environment that contains the body of two nested loops is:

env = [[MOO MOO OOO moo moo], [MOO OOO moo moo]]

Note that the desired effect could have been achieved using *continuations* (see the semantics for break in [Nielson92]). Still, the use of continuations still requires a *nesting* flag (see below), so that the semantics will be equally complex.

Loop execution

If $m_p \neq 0$, then all instructions following `MOO` must be executed until the matching `moo` is found. Since we cannot look back in the code after finding a `moo` instruction, we must use an environment. Upon entry of the loop, we save all the statements that follow `MOO` in the environment, and upon execution of `moo` we take them off again. This process can be executed with nesting, where multiple pieces of code are stored in the environment.

$$[\text{MOO}^{\text{loop}}] \quad \frac{\text{push(MOO S, env)} + \langle S, s\rangle \rightarrow s'}{\text{env} + \langle \text{MOO S}, s\rangle \rightarrow s'} \quad \text{if } m_p \neq 0$$

$$[\text{moo}^{\text{loop}}] \quad \frac{\text{pop(env)} + \langle S, s'\rangle \rightarrow s'}{\text{env} + \langle \text{moo}, s\rangle \rightarrow s'} \quad \text{where top(env)} = S$$

Access to the environment is provided using the functions *push*, *pop* and *top*, which are defined as follows:

push:	$S \times \text{env} \rightarrow \text{env}$	= env ++ S
pop:	$\text{env} \rightarrow \text{env}$	= tail(env)
top:	$\text{env} \rightarrow S$	= head(env)

The environment is a list of statement lists. The auxiliary functions ++, *head* and *tail* have definitions similiar to their definitions in the *Clean* language.

Forward search

According to the specification of `MOO`, this instruction must skip all instructions that follow it (until the matching `moo` is found) when $m_p = 0$. Note that when searching, we must always skip the instruction that directly following `MOO`.

In order to fulfill this last condition, we immediately replace `MOO` with `MOO2` and skip the instruction immediately following `MOO`. We use the composition of statements in the five natural semantics rules for `MOO` below. To make sure that searching only stops when the matching `moo` is found (and not just a nested moo), we keep track of the nesting level in the state (as *l*).

$$[\text{MOO}^{\text{begin search}}] \quad \frac{\text{env} + \langle \text{MOO2 } S_2, s\rangle \rightarrow s'}{\text{env} + \langle (\text{MOO } S_1)\ S_2, s\rangle \rightarrow s'} \quad \text{if } m_p = 0$$

$$[\text{MOO}^{\text{search}}] \quad \frac{\text{env} + \langle \text{MOO2 } S_2, s\rangle \rightarrow s'}{\text{env} + \langle (\text{MOO2 } S_1)\ S_2, s\rangle \rightarrow s'} \quad S_1 \neq \text{MOO} \ \&\ S_1 \neq \text{moo}$$

$$[\text{MOO}^{\text{begin nest}}] \quad \frac{\text{env} + \langle \text{MOO2 } S_2, s[l \rightarrow l + 1]\rangle \rightarrow s'}{}$$

$$[\mathrm{MOO}^{\text{end nest}}]\quad \frac{\text{env} + \langle(\mathrm{MOO2\ MOO})\ S_2, s\rangle \rightarrow s' \quad \text{env} + \langle \mathrm{MOO2}\ S_2, s[l \rightarrow l-1]\rangle \rightarrow s'}{\text{env} + \langle(\mathrm{MOO2\ moo})\ S_2, s\rangle \rightarrow s'} \quad \text{if } l > 0$$

$$[\mathrm{MOO}^{\text{end search}}]\quad \text{env} + \langle(\mathrm{MOO2\ moo})\ S_2, s\rangle \rightarrow s \quad \text{if } l = 0$$

Example of a nested loop

We determine the natural semantics of the following program, starting from a state where $m_p = 1$. The program consist of two nested loops, both of which must be executed exactly one (since $m_p = 1$).

```
MOO
OOM
MOO      } inner
OOO      } loop
moo      } outer
moo      } loop
```

$$\frac{}{[\,] + \langle \mathrm{MOO2\ moo}, s\rangle \rightarrow s}$$

$$\frac{}{[\,] + \langle \mathrm{MOO2\ moo}, s[l \rightarrow l-1]\rangle \rightarrow s}$$

$$\frac{}{[\,] + \langle \mathrm{MOO2\ moo\ moo}, s\rangle \rightarrow s}$$

$$\frac{}{[\,] + \langle \mathrm{MOO2\ OOO\ moo\ moo}, s[l \rightarrow l+1]\rangle \rightarrow s}$$

$$\frac{}{[\mathrm{outer}] + \langle \mathrm{MOO2\ moo\ moo}, s\rangle \rightarrow s} \qquad \frac{}{[\,] + \langle \mathrm{MOO2\ MOO\ OOO\ moo\ moo}, s\rangle \rightarrow s}$$

$$\frac{}{[\mathrm{outer}] + \langle \mathrm{MOO\ OOO\ moo\ moo}, s\rangle \rightarrow s} \qquad \frac{}{[\,] + \langle \mathrm{MOO\ OOM\ MOO\ OOO\ moo\ moo}, s\rangle \rightarrow s}$$

$$\frac{}{[\mathrm{outer,inner}] \langle \mathrm{moo}, s\rangle \rightarrow s} \qquad \frac{}{[\mathrm{outer}] \langle \mathrm{moo}, s\rangle \rightarrow s}$$

$$\frac{}{[\mathrm{outer,inner}] \langle \mathrm{moo\ moo}, s\rangle \rightarrow s}$$

$$\frac{}{[\mathrm{outer,inner}] + \langle \mathrm{OOO}, s\rangle \rightarrow s[m_p \rightarrow 0]}$$

$$\frac{}{[\mathrm{outer,inner}:=[\mathrm{MOO\ OOO\ moo\ moo}]] + \langle \mathrm{OOO\ moo\ moo}, s\rangle \rightarrow s}$$

$$\frac{}{[\mathrm{outer}] \langle \mathrm{MOO\ OOO\ moo\ moo}, s\rangle \rightarrow s}$$

$$\frac{}{[\mathrm{outer}] + \langle \mathrm{OOM}, s\rangle \rightarrow s[w \rightarrow w\ ++\ m_p]}$$

$$\frac{}{[\mathrm{outer}:=[\mathrm{MOO\ OOM\ MOO\ OOO\ moo\ moo}]] + \langle \mathrm{OOM\ MOO\ OOO\ moo\ moo}, s\rangle \rightarrow s}$$

$$\frac{}{[\,] + \langle \mathrm{MOO\ OOM\ MOO\ OOO\ moo\ moo}, s\rangle \rightarrow s}$$

Explanation

The first `MOO` in the program initiates a loop that will be executed. Immediately before execution, the loop body (i.e. the rest of the program) is stored in the environment and execution continues with the statement that follows `MOO`. The next `MOO` does the same thing (since m_p is still 1) and the remainder of the program after the second `MOO` is also placed in the environment. Execution continues.

The first `moo` makes sure that the inner loop loops. It retrieves the code for the inner loop from the stack and executes it. At this point $m_p = 0$, so that all instructions after `MOO` are skipped until the matching `moo` is found, after which execution is resumed. The same thing happens for the last `moo`, but note that after a `MOO` with $m_p = 0$, the instruction immediately following it is always skipped, until the *matching* `moo` is found. This means that we must keep track of the *nesting* level in the state (and we do just that in this example).

Instruction generation

The natural semantics described above still lacks the `mOO` instruction, which executes a value from memory as if it were an instruction. With our loop definitions this works just right. The natural semantics of `mOO` can de defined as a group of rules. There is no rule for $m_p = 3$, since that would result in an infinite loop.

$$[mOO^{0}]\ \frac{\langle moo, s\rangle \rightarrow s'}{\langle mOO, s\rangle \rightarrow s'}\ \text{if } m_p = 0$$

$$[mOO^{1}]\ \frac{\langle mOo, s\rangle \rightarrow s'}{\langle mOO, s\rangle \rightarrow s'}\ \text{if } m_p = 1$$

$$[mOO^{2}]\ \frac{\langle moO, s\rangle \rightarrow s'}{\langle mOO, s\rangle \rightarrow s'}\ \text{if } m_p = 2$$

$$[mOO^{4}]\ \frac{\langle Moo, s\rangle \rightarrow s'}{\langle mOO, s\rangle \rightarrow s'}\ \text{if } m_p = 4$$

$$[mOO^{5}]\ \frac{\langle MOo, s\rangle \rightarrow s'}{\langle mOO, s\rangle \rightarrow s'}\ \text{if } m_p = 5$$

$$[mOO^{6}]\ \frac{\langle MoO, s\rangle \rightarrow s'}{\langle mOO, s\rangle \rightarrow s'}\ \text{if } m_p = 6$$

$$[mOO^{7}]\ \frac{\langle MOO, s\rangle \rightarrow s'}{\langle mOO, s\rangle \rightarrow s'}\ \text{if } m_p = 7$$

$$[mOO^{8}]\ \frac{\langle OOO, s\rangle \rightarrow s'}{\langle mOO, s\rangle \rightarrow s'}\ \text{if } m_p = 8$$

$$[mOO^{9}]\ \frac{\langle MMM, s\rangle \rightarrow s'}{\langle mOO, s\rangle \rightarrow s'}\ \text{if } m_p = 9$$

$$[mOO^{10}]\ \frac{\langle OOM, s\rangle \rightarrow s'}{\langle mOO, s\rangle \rightarrow s'}\ \text{if } m_p = 10$$

$$[mOO^{11}]\ \frac{\langle oom, s\rangle \rightarrow s'}{\langle mOO, s\rangle \rightarrow s'}\ \text{if } m_p = 11$$

Structural Operational Semantics

As opposed to natural semantics (big-step semantics), structural operations (small-step semantics) describes how the first (smallest) step of the execution of a statement takes place.

Trivial instructions

The instructions that were trivial to specify in natural semantics (all instructions save 0, 3 and 7) are also trivial in structural operational semantics, since they consist of precisely one step. These instructions reduce to a final state right away. An exception is the composition of statements, which consists of two parts.

	[comp^1]	$\frac{<S_1, s> \Rightarrow <S'_1, s'>}{<S_1\ S_2, s> \Rightarrow <S'_1\ S_2, s'>}$		If statement S_1 cannot be executed ine one step
	[comp^2]	$\frac{<S_1, s> \Rightarrow s'}{<S_1\ S_2, s> \Rightarrow <S_2, s'>}$		If statement S_1 can be executed in one step
1	[mOo$^{p=0}$]	<mOo, s>	$\Rightarrow s$	if p = 0
2	[moO]	<moO, s>	$\Rightarrow s[p \rightarrow p + 1]$	
4	[Moo^{in}]	<Moo, s>	$\Rightarrow s[m_p \rightarrow v_0, v \rightarrow tail(v)]$	if $m_p = 0$
4	[Moo^{out}]	<Moo, s>	$\Rightarrow s[v \rightarrow v + m_p]$	if $m_p \neq 0$
5	[MOo]	<MOo, s>	$\Rightarrow s[m_p \rightarrow m_p - 1]$	
6	[MoO]	<MoO, s>	$\Rightarrow s[m_p \rightarrow m_p + 1]$	
8	[OOO]	<OOO, s>	$\Rightarrow s[m_p \rightarrow 0]$	
9	[MMM]	<MMM, s>	$\Rightarrow s[r \rightarrow m_p]$	if $r = \varepsilon$
9	[MMM]	<MMM, s>	$\Rightarrow s[m_p \rightarrow r, r \rightarrow \varepsilon]$	if $r \neq \varepsilon$
10	[OOM]	<OOM, s>	$\Rightarrow s[w \rightarrow w ++ m_p]$	
11	[oom]	<oom, s>	$\Rightarrow s[m_p \rightarrow v_0, v \rightarrow tail(v)]$	

Loop instructions

Just like in natural semantics, in structural operational semantics we use an enviroment to store the bodies of while loops. Because of the existence of 'half' while loops it was necessary to denote the natural semantics in a style which strongly resembles structural operational semantics, and it turns out that the denotation of the latter does not differ much from the former.

Loop execution

The rules for loop instructions are analogous to the natural semantics rules, the only difference being that we only rewrite the instructions, and do not describe an iteration's final state (i.e. there is no s').

[MOO^{loop}]	env + <MOO S, s> $\Rightarrow$ push(MOO S, env) + <S, s>	if $m_p \neq 0$
[moo^{loop}]	env + <moo, s> $\rightarrow$ pop(env) + <S, s>	where top(env) = S

The functions *push*, *pop* and *top* that allow access to the environment are unchanged.

Forward search

The rules for forward search in a loop are also analogous to the rules in natural semantics. The structural operational semantics is easier to read because we search through the loop step-by-step (something we could only do in natural semantics by jumping through some hoops).

[MOO$^{\text{begin search}}$]	env + <(MOO1 S_1) S_2, s> ⇒ env + <MOO2 S_2, s>	if $m_p = 0$
[MOO$^{\text{search}}$]	env + <(MOO2 S_1) S_2, s> ⇒ env + <MOO2 S_2, s>	$S_1 \neq$ MOO & $S_1 \neq$ moo
[MOO$^{\text{begin nest}}$]	env + <(MOO2 MOO) S_2, s> ⇒ env + <MOO2 S_2, s[l → l + 1]>	
[MOO$^{\text{end nest}}$]	env + <(MOO2 moo) S_2, s> ⇒ env + <MOO2 S_2, s[l → l – 1]>	if l > 0
[MOO$^{\text{end search}}$]	env + <(MOO2 moo) S_2, s> ⇒ <S_2, s>	if l = 0

Instruction generation

The rules for instruction generation with `m00` are the same as in natural semantics (we change → to ⇒). We do not include them here.

Comparison of semantics denotations

Now that we have specified the natural and structural operational semantics of COW, we are in a position to consider the differences. The natural semantics is useful if we are able to define a relationship between the initial state and the final state of the execution of an instruction (like we can do in the *While*-language [Nielson92]). In COW we were unable to do this for loops, due to the existence of 'half' while loops: there is no guarantee that a `M00` (begin loop) is followed by a matching `moo` (end loop). In the structural operational semantics we only look ahead one step, resulting in more readable semantics.

Turing Completeness

We still need to prove that the programming language COW is Turing complete. This can be proven in the following ways [Faase]:

1) Show that there is a mapping from each possible Turing machine to a program in COW;
2) Show that there is a COW program that simulates a universal Turing machine;
3) Show that COW is equivalent to a language known to be Turing complete;
4) Show that COW is able to compute all computable functions.

Options 2 and 3 look to be very complex. Option one is interesting, but will result in a large COW program that will serve as proof. Since COW programs are hard to read, we do not think that this is a convenient proof to follow, but the approach is clear: if we can show that COW is able to search through a table structure in memory, read coded Turing machine transitions from it and execute them, then we proven most of what we need to prove.

Computable functions

We believe that option 4 gives us the most transparent proof. According to [Sudkamp98] the computable functions are:

a) The functions
   - successor s: $s(x) = x + 1$
   - zero z: $z(x) = 0$
   - projection $p_i^{(n)}$: $p_i^{(n)} (x_1 \ldots x_n) = x_i$, $1 \leq i \leq n$

b) Functions created through functional composition, i.e.
   $f(x_1,\ldots, x_k) = h(g_1(x_1,\ldots, x_k), \ldots, g_2(x_1,\ldots, x_k))$
   The computability of f follows if g and h are computable.

c) Functions created through primitive recursion.
   If g and h are computable (with n and n+2 arguments, respectively), then
   $f (x_1,\ldots, x_n, 0) = g (x_1, \ldots, x_n)$
   $f (x_1,\ldots, x_n, y+1) = h (x_1, \ldots, x_n, y, f (x_1, \ldots, x_n, y))$
   is also computable.

d) The function f created through *minimalisation* is also computable:
   $f (x_1,\ldots, x_n) = \mu z [p(x_1,\ldots, x_n, z)]$
   if p is a computable predicate with n+1 arguments. Minimalisation yields the smallest z for which $p(x_1,\ldots, x_n, z)$ is true (1).

Proof structure

We need to show that

a) there is a function mechanism in COW;
b) s, z and p exist in COW;
c) functies in COW can be composed;
d) a COW function can call itself recursively;
e) minimalisatie can be implemented in COW (using recursion).

If COW can do all of these things, then the language is Turing complete.

Function mechanism

We state that

a) If COW is able to provide a function with its arguments, somewhere in memory (the 'stack') and

b) some piece of code (the function) can use these arguments and replace its first argument with its result after execution

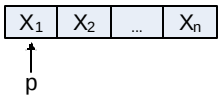
then we have a function mechanism. The nice thing about this mechanism is that functional composition follows immediately from it: if the function g in *f o* g yields x, then this result is placed in the memory location where f expects its argument.

We will say the a function call consists of a *prelude*, which sets up the arguments for use by a function f, and a *postlude*, which performs cleanup if necessary.

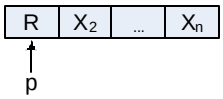
The *prelude* does the following:

- Moves the memory pointer to the memory area where the function arguments will be placed. This can be done using the instructions `m0o` and `mo0`.
- For each argument x_i:
  - Let m[p] := x_i. This can be done with `000` and `Mo0`.
  - If more arguments follow, the prelude increases the memory pointer by one (with `mo0`).
- Move the memory pointer back to the first argument.

The memory contents from memory pointer *p,* after the prelude, looks like this:



The function body follows the prelude, and this code may assume that the memory pointer points to its first argument. The function knows how many arguments it has, and provides the code that retrieves the arguments. The function is responsible for replacing its first argument with its result. The result after execution of a function is therefore:



Here, *R* represents the function result. Note that the values $x_2 \ldots x_n$ are undefined. The function is free to change them. It might be better if *R* did not overwrite a function argument (cf. the C function call mechanism), so that the prelude would have to reserve space for R on the stack. However, the approach we use is sufficient for our purposes.

The *postlude* has no work to do. With our mechanism, the code that follows *f* can use the function result, even if this code is another function *g* (with one argument). We could have a postlude that sets all arguments used (except the result) to zero, but this is unnecessary.

Basic functions

We show that the functions s, z and $p_i^{(n)}$ exist in COW by defining them.

The successor function is:

$s \equiv$ `MoO`

The zero function is:

$z \equiv$ `OOO`

The projection function is:

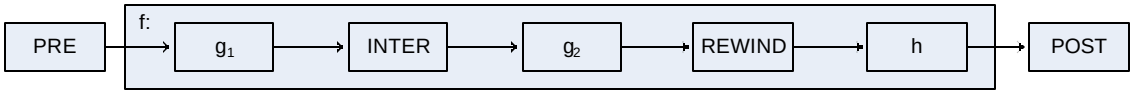
$p_i^{(n)} \equiv$ `moO`$^{i-1}$ `MMM` `mOo`$^{i-1}$ `MMM`

Functional composition

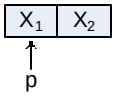
We define functional composition as:

> Let *f* be the composition of a function *h*: $\mathbb{N}^n \rightarrow \mathbb{N}$ with the functions $g_1, g_2, \ldots, g_n$, all $\mathbb{N}^k \rightarrow \mathbb{N}$. If all g_i and *h* are computable, then *f* is also computable.

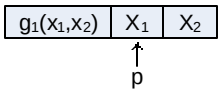
In COW, such a construction looks like this (for the sample function $h(g_1(x_1,x_2),g_2(x_1,x_2))$:



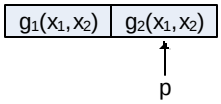
The prelude places the arguments for function g_1 on the stack, so that the stack looks like this: (for the sample function):



The function g_1 uses the arguments x_1 and x_2, and places its result at position *p*. We now place an *interlude* between two function g_i and g_j, which increases the memory pointer *p* by 1 and places the arguments for function g_j on the stack. The result of the interlude after g_1 in this example:



Execution of g_2 yields:



In order to execute *h* with the results of g_1 and g_2 we need the *rewind* element. This code moves the memory pointer back to the first result, i.e. the instruction `mOo` is executed $n-1$ times. After this, *h* can be executed directly with the arguments on the stack. So, *rewind* is the prelude of *h.*

It follows from this example that functional composition is possible in COW.

Primitive recursion

COW does not have a function *call* mechanism. The only tool we have to implement recursion is the while loop. We can do this by evaluating the recursion *bottom-up*.

Primitive recursion is defined as:

$f(x_1,\ldots,x_n,0) = g(x_1,\ldots,x_n)$
$f(x_1,\ldots,x_n,y+1) = h(x_1,\ldots,x_n,y,f(x_1,\ldots,x_n,y))$

If we *first* compute $g(x_1, \ldots, x_n)$, then apply the function *h* to the result (and a suitable *y*), and repeat the application of *h* as many times as necessary, then we achieve the desired result. Of course a recursion is always evaluated this way, but we make the order explicit in our proof, so that we can solve the recursion using a while loop.

Diagrams are too complex for this implementation. We will illustrate the approach by showing the contents of the stack during the computation, where $\underline{x}$ is short for the vector $x_1, x_2, \ldots, x_n$.

$\underline{x}$, y	The prelude of *f* places $\underline{x}$ and *y* on the stack.
$\underline{x}$, y, g($\underline{x}$)	$g(\underline{x})$ is evaluated. This is the basis for the recursion. A postlude makes sure that the result of $g(\underline{x})$ is stored on the stack just after the arguments $\underline{x}$ and *y*.
$\underline{x}$, y = 1, g($\underline{x}$), c	Now, let *c := y,* and *y := 1*. The counter *c* keeps track of the number of recursion iterations we must still execute. Since the counter decrements (rather than increments), we can use a while loop.
$\underline{x}$, 1, h($\underline{x}$,1,g($\underline{x}$)) c-1	We execute *h* with parameters $\underline{x}$, *y* and $g(\underline{x})$. Here, $g(\underline{x})$ is the value of *f* for $(\underline{x}, y-1)$.
$\underline{x}$, 2, h($\underline{x}$,2,h($\underline{x}$,1,g($\underline{x}$))) c-2	We execute *h* with parameters $\underline{x}$, *y* and $h(\underline{x},2,f)$, where *f* represents the previous value of the recursion.
$\underline{x}$, 3, h($\underline{x}$,3,h($\underline{x}$,2,h($\underline{x}$,1,g($\underline{x}$)))) c-3	We execute *h* with parameters $\underline{x}$, *y* and $h(\underline{x},3,f)$, where *f* represents the previous value of the recursion.
h($\underline{x}$, y, f($\underline{x}$, y))	When the counter *c* reaches 0, the recursion ends. This means that in our implementation, the while loop is not repeated anymore. A postlude makes sure that the result $h(\underline{x},y,f(\underline{x},y))$ is stored on the first stack position, so that it may be available for a next function in a functional composition.

In order to evaluate the computation suggested by this stack trace, COW must be able to

- Copy function arguments;
- Move function arguments;
- Execute a while loop.

COW has instructions to perform these operations (as already shown in the discussions on the function mechanism and functional composition), so that primitive recursion can be implemented in COW.

Minimalisation

Minimalisation also uses recursion, but the solution is simpler in this case. We make use of a stack trace to illustrate minimalisation:

$\underline{x}$	The prelude of function *f* places $\underline{x}$ on the stack.
1, $\underline{x}$	The termination condition of the minimalisation (is there a suitable *z* yet?) is initially untrue. We represent this as 1, since this value is used to decide whether the while loop should be executed again.
1, 0, $\underline{x}$, 0	We start with *z = 0*. A copy of *z* is placed on the second stack position (and also behind $\underline{x}$).
1, 0, p($\underline{x}$, 0) = 0	The predicate *p* is evaluated with arguments $\underline{x}$ and *z*. Suppose it yields 0.
1, 0, p($\underline{x}$, 0) = 0	The result is copied to the first stack position, and then

	inverted.
1, 1, x, 1	The while loop repeats. *z* is incremented by 1 and placed on the stack. We needed the copy of *z* to determine previous value.
1, 1, p(x, 1) = 1	The predicate *p* is evaluated with arguments x and *z*. Suppose it yields 1.
0, 1, p(x, 1) = 1	The result is copied to the first stack position, then inverted. The while loop now terminatesm because the termination condition (the first stack position) is now 0.
1, …	The current value of *z* is copied to the first stack position and represents the result of the minimalisation.

It follows that we can realize minimalisation in COW using a while loop.

Result

We have shown that COW has a function mechanism, with which we can implement the successor function, the zero function, the projecion function, functional composition, primitive recursion and minimalisation. From this we may deduce that COW is Turing complete.

References

[Faase] Faase, Frans: *Brainf*** *is Turing-Complete*
`http://home.planet.nl/~faase009/Ha_bf_Turing.html`

[Heber] Heber, Sean: COW – *Programming for Bovines*
`http://www.bigzaphod.org/cow/`

[Nielson92] Nielson, Hanne Riis & Nielson, Flemming: *Semantics with Applications: a Formal Introduction*, John Wiley & Sons, 1992.
`http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.pdf`

[Sudkamp98] Sudkamp, Thomas: *Languages and Machines*, 2nd edition, Addison Wesley Longman, 1998.