From Turing machines to computer viruses

BY JEAN-YVES MARION*

Université de Lorraine, LORIA, UMR 7503, Vandoeuvre-lès-Nancy
F-54500, France

Self-replication is one of the fundamental aspects of computing where a program or a system may duplicate, evolve and mutate. Our point of view is that Kleene’s (second) recursion theorem is essential to understand self-replication mechanisms. An interesting example of self-replication codes is given by computer viruses. This was initially explained in the seminal works of Cohen and of Adleman in the 1980s. In fact, the different variants of recursion theorems provide and explain constructions of self-replicating codes and, as a result, of various classes of malware. None of the results are new from the point of view of computability theory. We now propose a self-modifying register machine as a model of computation in which we can effectively deal with the self-reproduction and in which new offsprings can be activated as independent organisms.

Keywords: virus; recursion theorem; universal Turing machine; stored register machine; self-reproduction; auto-modification

A virus is a virus!

André Lwoff, 1959

1. Introduction

(a) From Turing machines to computer viruses

The Turing model of computation (MOC) is the bare bone of computability and complexity. Turing machines (TMs) are simple and efficient. A TM consists of a control unit, which is a hard-wired internal program, and of one or several tapes. In his original paper, Turing [1,2] designed a universal Turing machine (UTM), which corresponds nowadays to a self-interpreter. A UTM is able to interpret any Turing machine (TM) from an encoding of it. In particular, a UTM is able to interpret itself. Turing’s result suggested that a machine can have access to its own code, that is, to a self-description. The second recursion theorem of Kleene [3] allows us to understand the notion of self-reference. His demonstration suggests the description of a self-reproducing machine. Moschovakis [4] conducted a survey on the applications of this little gem. The reader may also consult the books of Smullyan [5,6].

About 50 years later, Cohen [7] defined computer viruses: ‘A virus is a program that is able to infect other programs by modifying them to include a possibly evolved copy to itself’. More recently, Cohen’s definition was rephrased by Ször [8]

*jean-yves.marion@loria.fr

One contribution of 18 to a Theme Issue ‘The foundations of computation, physics and mentality: the Turing legacy’.
in his book. Cohen’s definition seems to be accepted by the computer security community. There are only a few works on theoretical aspects of computer viruses (e.g. [9–12]).

(i) What are we studying and what we will not consider?

Because there is a consensus on the definition of Cohen and of Adleman [9], it will be our starting point. Therefore, we will focus on the features of computer viruses related to self-reproduction, to code mutation and to self-modification. Thus, and for this study, a computer virus is defined as a self-replicating program, whose offspring may be a mutation of the original program. Moreover, the offspring may be activated, implying a new generation and so the contamination can go on. In spite of widespread works on malware, there are only a few theoretical studies on computer viruses. This study is not about the practical side of viruses. For this, the reader may refer, for example, to Ször’s book [8].

We are going to illustrate our approach by a first example. A more complete example will be given in the last part of the introduction. Consider a simple virus, say VA, that tries to contaminate each running process in a system. For this, VA checks first whether or not a process is infected. Thus, it avoids superinfection and, in particular, it avoids infecting itself when executed. If the process is sane, i.e. if the process has not been already infected by the virus VA, then it injects a copy of itself. The specification of the virus is written in pseudo-code as follows:

\[
\text{Specification\_of\_VA}(v) \\
\text{For each running process } p \\
\text{If } p \text{ is not infected then inject } v \text{ to } p.
\]

The parameter \( v \) stands for a self-description of VA. To design VA, we have to construct a program \( v \) such that, roughly speaking, \( v = \text{Specification\_of\_VA}(v) \). In other words, \( v \) is a fixpoint of the specification of VA. This example illustrates the fact that Cohen’s or Adleman’s model bears a resemblance to the second recursion theorem of Kleene [3] and to von Neumann [13] self-reproducing machines.

(ii) A complete model of self-reproduction and main objectives

Before going further, it is worth defining three concepts that are essential for this study. Self-reference is the ability of a program to refer to itself. A recursive algorithm is a typical example of self-reference in programming language. In this case, a program has a pointer referring to its own code. Self-replication is the ability of a program to copy itself. A well-known example is quines. The duplication process may produce a different program (the offspring) from the original one (the father’s offspring). Indeed, there could be a mutation of the original program. In this case, the offspring may or may not compute the same thing as the original program. Self-modification is the ability of a program to modify itself. A self-modifying program has the ability to alter its own code. For example, it may change its instructions on the fly or it may generate codes from data that it can access, and thus the overall program may be seen as a sequence of different code layers. These three notions are distinct. A programming language can be self-referential without having the ability to alter its code. These three
notions correspond to the notion of self-knowledge developed in Case [14]. We do think that a complete model of self-reproducing systems must reflect all these three notions and also the important fact that a copy can be activated.

The main objective of this study was to propose a synthetic model of computer viruses and more generally of complete self-reproduction. From a programming perspective, this means a model in which computer viruses are generated from a viral specification similar to the one given previously. For these reasons, we focus on effective models of computation from the theory of programming languages, computability and complexity theory. We propose an MOC that explains, intuitively, the essential ideas behind self-reproduction with their connections to recursion theorems. This MOC is a simple model of self-reproduction and is complete in the sense that we have just explained. Moreover, it might be seen as a basic model to study self-modifying programs.

(iii) *A short tour in the land of model of computation*

Seminal models, such as TMs or Post [15] tag systems, process strings. Shepherdson & Sturgis [16] seem to be the first to introduce register machines over words. More recently, Moss [17] proposes text register machines (TRMs) in order to have a simple and elegant computation model in which constructions in computability theory are implemented. The TRM MOC can manipulate a program as data.

We present a MOC named a self-modifying register machine (SRM), which is an extension of TRMs. SRMs work over a finite alphabet. Programs are stored in registers and can be processed as data. The crucial difference with TRMs is that any register may be activated and run. As a result, a program may generate a copy of itself, and then run it. Thus, the SRMs seem to be an adequate model of self-reproducing systems.

An SRM is a stored program abstract machine. As in current computer architectures, a program is loaded into memory. It may modify itself either by changing its instructions or by executing instructions outside the original program. Turing & Wilkinson [18] with the Automatic Computing Engine project and Burks et al. [19] were the first main protagonists to conceive such architectures. Elgot & Robinson [20], Hartmanis [21] and Cook & Reckhow [22] defined random access stored program machines, RASPs, in a similar way. Unlike RASPs, SRMs have no indirect jumps, which is a questionable feature because it tends to ignore the computing locality principle.

(iv) *Self-reproducing Turing machines*

UTMs and the second recursion theorem form the core of the study of self-reproducing machines. Later, in his paper on morphogenesis, Turing [23] studied a mathematical model of growing and evolving organisms. At the same time, von Neumann & Burks [24] constructed a self-replicating cellular automaton. The clear intention of these works was to have synthetic models of living organisms. More than 50 years later, the theory of self-reproduction was extended in different directions, as described by Sipper [25]. The relations between constructions within the von Neumann model and recursion theorem are quite well known, and are well explained by Case [26] and Arbib [27], for example. Within the Turing MOC, it is essentially Lee [28] and Thatcher [29] who worked on self-describing TMs,
and more recently Margenstern & Rogozhin [30]. In essence, the design of a self-describing machine follows Rogers [31], which is based on the recursion theorem proof: suppose that \( \text{Diag}(z) \) is the code of the TM \( M_z(z) \) (where, for any \( z \), \( M_z \) is the machine with code number \( z \)). There is a code \( q \) such that for all \( d \), 
\[
M_q(d) = \text{Diag}(d).
\]
Hence, by taking \( \text{Diag}(q) \) for \( e \), we obtain for all \( d \) 
\[
M_e(d) = M_{\text{Diag}(q)}(d) = M_q(q) = \text{Diag}(q) = e.
\]
The machine of code \( e \) duplicates itself. This mechanism is behind all further constructions that we will see.

(b) *A compiler virus*

We now turn to a second example. In his Turing award lecture, Thompson [32] gives a construction of a *compiler virus*. We will use Thompson’s construction to illustrate that mechanisms, underlying computer viruses, depend directly on fundamental theorems of computability.

We are now given a self-interpreter \( \text{interp} \). For any program \( p \), a self-interpreter computes \( \text{interp}(p) = [p] \). We write \( [p] \) to denote the function that is computed by the program \( p \).

Say that an *interpreter virus* is a virus, which infects an interpreter. Then, the infected interpreter may compromise the program it runs. That is, an interpreter virus, say \( v \), transforms \( \text{interp} \) into an infected interpreter \( \text{interp}^v = \text{interp}^v(p) = [v](\text{interp}) \). Thus, if we replace the trusted interpreter \( \text{interp} \) by \( [v](\text{interp}) \), instead of running a program \( p \), we will run \( [v](\text{interp}) \).

The program of \( [v](\text{interp}) \) is easily built, thanks to the s-m-n theorem. The concept behind the s-m-n function is that of program specializers. A program specializer \( \text{spec} \) is a program transformer that takes two inputs: a program \( p \) and data \( c \). The result \( \text{spec}(p,c) \) is a program such that \( \text{interp}([\text{spec}(p,c)](d)) = [p](c,d) \). As a result, a program that computes \( [v](\text{interp}) \) is \( \text{spec}(v,\text{interp}) \). Indeed, we have \( \text{interp}([\text{spec}(v,\text{interp})](p)) = [v](\text{interp}) \). So, \( \text{spec}(v,\text{interp}) \) is the result of the infection of \( \text{interp} \) by the virus \( v \).

From there, the virus may also inject itself into \( p \). For this, \( v \) uses a propagation vector in order to infect \( p \). In accordance with Bonfante et al. [11], the propagation vector is represented by a specializer \( \text{spec} \). For example, \( v \) may insert its program at the beginning or at the end of \( p \), or yet it may shuffle its code inside \( p \). However, it is not our real concern in this study.

Thus, the infection caused by the infected interpreter executes \( \text{spec}(v,p) \) instead of the same program \( p \). The problem is to show that there must exist a virus \( v \) that satisfies the following equation:
\[
[[\text{spec}(v,\text{interp})](p) = [\text{spec}(v,p)]. \tag{1.1}
\]
The solution of this problem consists of finding a fixpoint \( v \). This can be obtained by the second recursion theorem of Kleene [3].

Following Jones [33], we can generate a compiler from an interpreter and a specializer. A compiler \( \text{comp} \) is defined by 
\[
[\text{comp}](p) = [\text{spec}](\text{interp},p), \tag{1.2}
\]
and it is implemented by a self-application of specializers,
\[
\text{comp} = [\text{spec}](\text{spec}, \text{interp}).
\]  
(1.3)

Now, if the interpreter is compromised, then the new compiler \(\text{comp}'\), after recompilation, will be also compromised,
\[
[\text{comp}'](p) = [\text{spec}](\text{comp}([\text{spec}](v, \text{interp}), p),
\]  
(1.4)
where we recall that \(\text{comp}([\text{spec}](v, \text{interp}))\) is the code of the infected interpreter, and
\[
\text{comp}' = [\text{spec}](\text{spec}, \text{comp}([\text{spec}](v, \text{interp}))).
\]  
(1.5)
As a result, the compromised compiler will insert the virus into each new compiled program. The construction that we have presented follows the main lines of the design of Thompson’s *Trojan horse*. The theoretical grounds of this construction are the existence of a UTM, of the s-m-n functions and of program fixpoints. The aim of the remaining paper is to explain all these connections in more detail.

2. Self-modifying register machines

(a) Self-modifying register machines in a nutshell

The central notion of this study is that of SRMs, which allows us to study the diverse notions of self-reproduction. SRMs are based on the definition of TRMs suggested by Moss [17]. TRMs work on words of the alphabet \{1, #\}. Programs and data are words over \{1, #\}. An important feature of Moss TRMs is that the composition of programs is performed by a mere concatenation. Thus, there are no tedious coding devices. As a result, it is a fascinating exercise to explore correspondence between programming constructions and demonstrations of classical computability theory, such as the design of a program that computes program fixpoints. The elegance of this work is on the lineage of Jones [33].

An SRM is a text-register-stored program machine. There are two main differences with TRMs. First, programs are stored in registers, and second, any register can be executed. As a result, an SRM program is able to produce another program and to execute it. An SRM has a finite number of registers \(R_0, R_1, \ldots, R_m\), a *register pointer* \(RP\) and an *instruction pointer* \(IP\). A register \(R_n\) contains a word that may be decoded as a sequence of instructions. The program counter is broken into a register pointer and an instruction pointer. Hence, the current instruction is at address \(IP\) of the register pointed by \(RP\), which we denote \(R_{RP}\). For every instruction, the first step is to fetch and to execute the current instruction. The next instruction depends on the instruction kind. The instruction pointer \(IP\) then moves accordingly. The execution flow may move from one register to another by changing the values of the register pointer. One might think of an SRM as an abstract machine where each register contains a process. A single process is running at any time and a process may activate another one. Thus, any register can be activated and run. In the SRM model, there is no distinction between programs and data. Both are manipulated during a computation. So, programs are ‘first-class citizens’ because they are represented by data, whereas data are
Table 1. SRM instruction set where $\ell^n$ is $\ell$ concatenated $n$ times, $\ell \in \{1, \#\}$.

<table>
<thead>
<tr>
<th>instruction</th>
<th>opcode</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 put $n, 1$</td>
<td>$1^{n+1}#$</td>
<td>add 1 at the end of $R_{RP+n}$</td>
</tr>
<tr>
<td>2 put $n, #$</td>
<td>$1^{n+1}##$</td>
<td>add $#$ at the end of $R_{RP+n}$</td>
</tr>
<tr>
<td>3 jmp $+n$</td>
<td>$1^{n+1}###$</td>
<td>go forward $n$ steps</td>
</tr>
<tr>
<td>4 jmp $-n$</td>
<td>$1^{n+1}####$</td>
<td>go backward $n$ steps</td>
</tr>
<tr>
<td>5 case $n$</td>
<td>$1^{n+1}#####$</td>
<td>conditional jump depending on $R_{RP+n}$</td>
</tr>
<tr>
<td>6 exec $n$</td>
<td>$1^{n+1}######$</td>
<td>control transfer to register $R_{RP+n}$ and $IP = 0$</td>
</tr>
</tbody>
</table>

also ‘first-class citizens’ because they can be effectively executed. This last feature
is shared with RASPs, but not with the other models where a universal machine
or an interpreter is necessary.

(b) Design of the self-modifying register machines

Each register holds a word of the two symbol alphabet $\{1, \#\}$. Registers are
seen as unbounded first in, first out queues. That is, we pop the first letter to
read it, and we write a letter by adding it at the bottom of the register. Program
instructions are described in Table 1.

The program is initially stored in register $R_0$. In an execution of an SRM, the
code can move. The code motion is controlled by two features. The first feature is
the ability to copy and move data from one register to another one. The second
feature is the ability to activate a program in a register, owing to the instruction
exec. A program inside a register may generate a new program in another and
run it. As a result, the overall program, i.e. the set of instructions executed, is
spread over several registers.

All instructions except exec increment $IP$ and leave $RP$ unchanged. Jumps
(jmp) and conditional branches (case) are performed inside a register, and
addresses are relative to the current instruction ($IP$). An important decision in
the design of SRMs is the fact that register addressing is relative to the register
pointer $RP$. The register address consists of two parts: $RP$ and an offset $n$. This
means that the instruction put $1, n$ appends 1 at the bottom of the register
$R_{RP+n}$. Similarly, the instruction exec executes the instruction on the top (i.e.
$IP$ is initialized to 0) of the register $R_{RP+n}$. This mapping allows us to load the
program into any location, by a right translation. To sum up, a program loaded
in a register is ‘right handed’; that is, it manipulates and runs registers only on
its right. In return, a program has no access to registers on its left. In this way,
we clearly simplify program relocation. This feature will be taken into account
by the sequential semantics introduced later.

Discussion. This design allows us to express the fact that an offspring can
be created and activated. However, the offspring is independent and it cannot
‘communicate’ with its father. One might think of a different instruction set
or architecture. For example, we may have a relative addressing for registers
with positive and negative addresses. Thus, an instruction such as put $\ell, -n$
would mean that we add $\ell$ to register $R_{RP-n}$. We could also have exec $-n$
and exec $+n$ in order to run registers on the left and on the right of the current
program. It is worth noticing that these architectural choices have consequences
for fixpoint constructions that we will not explore in this work. Here, we choose simplicity. Finally, note that the overall address space also has two dimensions: one for instructions and one for registers.

(c) Semantics of the self-modifying register machine language

A small step semantics in the spirit of structural operational semantics of Plotkin [34] or of natural semantics [35] is now given. A configuration \((\mathcal{A}, \mathcal{RP}, n)\) consists of a store \(\mathcal{A}\), a register pointer \(\mathcal{RP}\) and a value \(n\), which is either the instruction pointer \(\mathcal{IP}\) or \(\text{Halt}\). In the last case, we say the configuration is final.

A store \(\mathcal{A}\) is a finite mapping such that \(\mathcal{A}(m) = R_m\) for all \(m\). The notation \(\mathcal{A}[R_m \leftarrow d]\), where \(d\) in \((1, \#)^*\) denotes the store \(\mathcal{A}'\) such that \(\mathcal{A}'(n) = R_n\) for all \(n\) except \(m\) where \(\mathcal{A}'(m) = d\). Actually, the notation \(\mathcal{A}\) is a convenient shortcut that we will eventually abandon to the gain of a more practical sequential semantics.

The current program is the program indicated by \(\mathcal{RP}\). Initially, the current program is in \(R_0\). We write \(\mathcal{RP}[\mathcal{IP}]\) the instruction that is pointed by \(\mathcal{IP}\) in the current program, i.e. the instruction at the address \(\mathcal{IP}\) in the register \(R_{\mathcal{RP}}\). Each instruction has an opcode, which is a word over \((1, \#)\). The program code is a word over \((1, \#)\), which consists of the concatenation of each instruction opcode composing the program. We write \(p \cdot q\) to mean the concatenation of both words \(p\) and \(q\). The size of the instruction encoding is variable, and pointer arithmetic works as usual. So, we write \(\mathcal{IP} = \mathcal{IP} + 1\) to increment the register \(\mathcal{IP}\) of the length of an instruction opcode, i.e. to move forwards of one instruction, and \(\mathcal{IP} = \mathcal{IP} - 1\) to decrement the register \(\mathcal{IP}\) of one instruction, i.e. to move backwards of one instruction. The exact address is obtained by determining the size of opcodes, which is easily calculated because the set of well-formed program opcodes is regular.

We define the single step transition as follows. \((\mathcal{A}, \mathcal{RP}, \mathcal{IP}) \rightarrow (\mathcal{A}', \mathcal{RP}', \mathcal{IP}')\) holds, if the execution of the instruction \(\mathcal{RP}[\mathcal{IP}]\) gives the store \(\mathcal{A}'\), following the rules of Table 2. The next instruction is then \(\mathcal{RP}'[\mathcal{IP}']\).

A computation is defined by the reflexive and transitive closure of \(\rightarrow\), and is noted \(\rightarrow^{*}\). At the beginning of the computation, the program is loaded into the register \(R_0\). Both \(\mathcal{RP}\) and \(\mathcal{IP}\) are both initially equal to 0. That is, \(\mathcal{IP}\) points to the first instruction contained in \(R_0\).

\[\text{Table 2. Small-step semantics, where } R_{\mathcal{RP}+n} = \mathcal{A}(\mathcal{RP} + n).\]

<table>
<thead>
<tr>
<th>Case</th>
<th>Store</th>
<th>Instruction</th>
<th>Effect</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>put (n, 1)</td>
<td>(\mathcal{A}, \mathcal{RP}, \mathcal{IP})</td>
<td>((\mathcal{A}[\mathcal{RP}+n \leftarrow \mathcal{RP}+n \cdot 1], \mathcal{RP}, \mathcal{IP} + 1))</td>
<td>if (\mathcal{IP} = 1^{n+1}#)</td>
<td></td>
</tr>
<tr>
<td>put (n, #)</td>
<td>(\mathcal{A}, \mathcal{RP}, \mathcal{IP})</td>
<td>((\mathcal{A}[\mathcal{RP}+n \leftarrow \mathcal{RP}+n \cdot #], \mathcal{RP}, \mathcal{IP} + 1))</td>
<td>if (\mathcal{IP} = 1^{n+1}#^2)</td>
<td></td>
</tr>
<tr>
<td>jmp +(n)</td>
<td>(\mathcal{A}, \mathcal{RP}, \mathcal{IP})</td>
<td>((\mathcal{A}, \mathcal{RP}, \mathcal{IP} + n))</td>
<td>if (\mathcal{IP} = 1^{n+1}#^3)</td>
<td></td>
</tr>
<tr>
<td>jmp -(n)</td>
<td>(\mathcal{A}, \mathcal{RP}, \mathcal{IP})</td>
<td>((\mathcal{A}, \mathcal{RP}, \mathcal{IP} - n))</td>
<td>if (\mathcal{IP} = 1^{n+1}#^4)</td>
<td></td>
</tr>
<tr>
<td>case (n)</td>
<td>(\mathcal{A}, \mathcal{RP}, \mathcal{IP})</td>
<td>((\mathcal{A}, \mathcal{RP}, \mathcal{IP} + 1))</td>
<td>if (\mathcal{IP} = 1^{n+1}#^5)</td>
<td></td>
</tr>
<tr>
<td>exec (n)</td>
<td>(\mathcal{A}, \mathcal{RP}, \mathcal{IP})</td>
<td>((\mathcal{A}, \mathcal{RP} + n, 0))</td>
<td>if (\mathcal{IP} = 1^{n+1}#^6)</td>
<td></td>
</tr>
<tr>
<td>otherwise</td>
<td>(\mathcal{A}, \mathcal{RP}, \mathcal{IP})</td>
<td>((\mathcal{A}, \mathcal{RP}, \text{Halt}))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Phil. Trans. R. Soc. A (2012)
Typically, the program halts if \( \text{IP} \) ‘goes below’ the current program. There are other reasons to halt. The register \( \text{IP} \) may point on a word, which is not the opcode of an instruction. Another way to halt is when a jump is performed with an address outside the register.

\[(d)\] Sequential semantics

We now introduce the notion of sequential semantics, which provides an easy way to express SRM computations. We write \((d_0, \ldots, d_i \bullet p, \ldots, d_n)\) to mean the configuration \((R, R_P, \text{IP})\), where \(R_P = i\), \(\text{IP}\) points to the first letter of \(p\) and \(R(j) = d_j\) for \(j = 0, n\) except for \(j = i\), where \(R(i) = d_i \bullet p\) and for \(j > n\), \(R(j) = e\). When no ambiguity can arise, we also call \((d_0, \ldots, d_i \bullet p, \ldots, d_n)\) a configuration.

The intuition is that the program denoted by \(d_i\) has already been executed, and we begin to run its sequel \(p\). So, the underlined word is the current executed program. In particular, \((d_0, \ldots, p, \ldots, d_n)\) means that \(\text{IP}\) points to the first letter of \(p\), and so we run \(p\). The initial configuration corresponds to \((p, d_1, \ldots, d_n)\).

Next, a computation is now written as
\[d_0, \ldots, d_i \bullet p, \ldots, d_n \Rightarrow d'_0, \ldots, d'_m\]
to say that if the SRM runs the program \(p\), then the computation goes to the configuration \((d'_0, \ldots, d'_m)\). Sometimes, no words will be underlined on the right, e.g.
\[d_0, \ldots, d_i \bullet \overline{p}, \ldots, d_n \Rightarrow d'_0, \ldots, d'_m,\]
In this case, this means that the computation halts. Sequential semantics will be used throughout the remainder of the paper. Examples in §2e illustrate how it works and allows us to get familiar with it.

\[(e)\] Program motions

In §2b, we saw that data contained in registers on the left of the current program are useless to the ongoing computation. Indeed, there is no instruction that can use registers on the left of the current program, that is on the left of \(R_{RP}\). We will state this property and use it as a shortcut to display computations in the remain.

**Proposition 2.1.**

— For any \(p\) if
\[c_0, \ldots, c_i, \overline{p}, d_1, \ldots, d_n \Rightarrow c'_0, \ldots, c'_j, p', d'_1, \ldots, d'_m,\]
then \(i = j\) and \(c_k = c'_k\) for any \(0 \leq k \leq i\).

— For any \(i, j\) and \(p\) if
\[c_0, \ldots, c_i, \overline{p}, d_1, \ldots, d_n \Rightarrow c_0, \ldots, c_i, p', d'_1, \ldots, d'_m,\]
then \(c'_0, \ldots, c'_j, p, d_1, \ldots, d_n \Rightarrow c'_0, \ldots, c'_j, p', d'_1, \ldots, d'_m,\)

This proposition means that the registers on the left of \(p\) may be forgotten. For this reason, we will tend not to write registers on the left side of the current

*Phil. Trans. R. Soc. A (2012)*
From Turing machines to computer viruses

program. When we want to drop the left part of the current program $p$, we will write

$$c_0, \ldots, c_i, p, d_1, \ldots, d_n \Rightarrow p, d_1, \ldots, d_n.$$  \hspace{1cm} (2.1)

Another important consequence is that we can relocate a program by translating it on the right. The result of its computation will be exactly the same. This means that if

$$p, d_1, \ldots, d_n \Rightarrow p', d_1', \ldots, d_m',$$

then for any $i$ and all $c_0, \ldots, c_i$,

$$c_0, \ldots, c_i, p, d_1, \ldots, d_n \Rightarrow c_0, \ldots, c_i, p', d_1', \ldots, d_m'.$$

In the remainder of this paper, we will use the statement (2.1) either as a simplification rule or as a program relocation rule.

(f) Sequential connectivity and universality

Now, we come to the notion of composition, which says that for any $p, q$ and $d_1, \ldots, d_n$, if

$$p, d_1, \ldots, d_n \Rightarrow p, d_1', \ldots, d_m',$$

and, in the above equation, $RP$ is unchanged and $IP$ points to the last instruction of $p$, and if

$$q, d_1', \ldots, d_m' \Rightarrow q, d_1'', \ldots, d_k'',$$

then

$$p \cdot q, d_1, \ldots, d_n \Rightarrow p \cdot q, d_1'', \ldots, d_k''.$$  

In the SRM framework, we prefer to refer to this compositional condition as a sequential connectivity condition. Indeed, it really suggests the idea that we have connected two computations. SRMs offer other connectivity mechanisms that we will not explore in this study.

It is evident that the SRM MOC is Turing complete. Nevertheless, it is worth seeing how universality boils down. A self-interpreter is a program $\text{interp}$ that satisfies

$$\text{interp}, p, d_1, \ldots, d_n \Rightarrow \underline{p}, d_1, \ldots, d_n.$$  

This equation holds because we can translate a program to the right. This is a consequence of equation (2.1). A very simple SRM self-interpreter is $\text{exec 1}$,

$$\text{exec 1}, p, d_1, \ldots, d_n \Rightarrow \underline{p}, d_1, \ldots, d_n.$$  

3. Illustrations

The following examples illustrate how to program SRMs. They will be used in constructions in the remainder. Note that programs are implicitly parametrized by a number $n$, which is an upper bound on the number of saved registers (or arguments). Thus, the reading is still rigorous, but notations are uncluttered. Registers beyond $n$ will be used as working registers to perform intermediate steps in a computation. We will suppose that they are empty. Lastly, to reduce notational overhead, we henceforth confuse and use the same notation for a program and its opcode.

Phil. Trans. R. Soc. A (2012)
(a) Move

The program move $R_i, R_j$ moves the contents of the register $R_i$ into the register $R_j$ and erases the content of $R_i$. We can easily express the sequential semantics of move $R_i, R_j$,

$$\text{move } R_i, R_j, d_1, \ldots, d_i, \ldots, d_j, \ldots d_n \Rightarrow \text{move } R_i, R_j, d_1, \ldots, \varepsilon, \ldots, d_j \bullet d_i, \ldots d_n.$$  

(3.1)

The program of move $R_i, R_j$ is the following:

1. case $i$
2. jmp 6 (* if empty *)
3. jmp 3 (* if $R_i$ begins with # *)
4. put $j,1$ (* insert 1 into $R_j$ *)
5. jmp $-4$ (* return at the first instruction *)
6. put $j,#$ (* insert # into $R_j$ *)
7. jmp $-6$ (* return at the first instruction *).

(b) Erase

Next, we define the program erase $R_j$, which erases the content of $R_j$,

1. case $j$
2. jmp 3 (* if empty *)
3. jmp $-2$ (* erase # *)
4. jmp $-3$ (* erase 1 *)

$$\text{erase } R_j, d_1, \ldots, d_j, \ldots, d_n \Rightarrow \text{erase } R_j, d_1, \ldots, \varepsilon, \ldots, d_n.$$  

(3.2)

(c) Shift

We now define a program shift$_n$ that shifts to the right the registers $R_1, \ldots, R_n$,

$$\text{shift}_n, d_1, \ldots, d_n \Rightarrow \text{shift}_n, \varepsilon, d_1, \ldots, d_n.$$  

(3.3)

We implement shift$_n$ as follows:

move $R_n, R_{n+1} \bullet \ldots \bullet \text{move } R_1, R_2 \bullet \text{erase } R_1$.

(d) Copy

The program $\text{cp } R_i, R_j$ copies the content of the register $R_i$ into the register $R_j$. Unlike move $R_i, R_j$, the register $R_i$ is not erased,

$$\text{cp } R_i, R_j, d_1, \ldots, d_i, \ldots, d_j, \ldots d_n \Rightarrow \text{cp } R_i, R_j, d_1, \ldots, d_i, \ldots, d_i, \ldots d_n.$$  

(3.4)
Hence, there are two copies of the same value at two different places. For this, we use $R_{n+1}$ that we suppose is empty. The program of $cp R_i, R_j$ is the following:

1. erase $R_j$
2. case $i$
3. jmp 8 (* if empty *)
4. jmp 4 (* if $R_1$ begins with # *)
5. put $j,1$ (* insert 1 into $R_j$ *)
6. put $n+1,1$ (* insert 1 into $R_{n+1}$ *)
7. jmp $-5$ (* return at the first instruction *)
8. put $j,#$ (* insert # into $R_j$ *)
9. put $n+1,#$ (* insert # into $R_{n+1}$ *)
10. jmp $-8$ (* return at the case instruction *)
11. move $R_{n+1},R_i$.

4. Kleene’s recursion theorem

We now come to fixpoint constructions in the SRM model. There are various fixpoint constructions induced by demonstrations of the recursion theorem of Kleene and of its variants. We will focus on a uniform version of the recursion theorem due to Myhill [36], according to Smullyan [6].

There is a fixpoint compiling function $fix$ such that for all $p \in \{1,\#\}^*$ and all $d_1, \ldots, d_n \in \{1,\#\}^*$,

$$fix(p), d_1, \ldots, d_n \Rightarrow p, fix(p), d_1, \ldots, d_n,$$

where the program $p$ can be interpreted as a specification to perform a computation with a self-copy. The function $fix$ generates a fixpoint of the program $p$. That is, it creates a copy of $p$, and then it runs the copy of $p$ on $fix(p)$. So, the computation goes on.

Actually, the function $fix$ is called a program transformation, which is a central notion in this part. More generally, a program transformation is a function, say $\Psi$, that takes a program $p$ (encoded by a word in $\{1,\#\}^*$), and possibly other inputs $d_1, \ldots, d_n$, and constructs a new program $\Psi(p, d_1, \ldots, d_n)$ (encoded by a word in $\{1,\#\}^*$). A compiler, or an s-m-n function, is a well-known example of program transformation (see Jones [33]). In an SRM model, a program transformation is implemented by a program, say $e$, that satisfies an equation of the form

$$e, p, d_1, \ldots, d_n \Rightarrow e, \Psi(p, d_1, \ldots, d_n), d_1, \ldots, d_n.$$

The result $\Psi(p, d_1, \ldots, d_n)$ is a program that can be activated in turn in the SRM model.
For example, recall from the introduction that a program specializer $\text{spec}$ implements an $s$-$m$-$n$ function. In an SRM model, $\lbrack \text{spec} \rbrack$ is a program transformation implemented by $\text{spec}$ satisfying

$$\lbrack \text{spec} \rbrack(p, d), d_1, \ldots, d_n \Rightarrow p, d, d_1, \ldots, d_n.$$ 

Note that $\lbrack \text{spec} \rbrack(p, d)$ activates the run of $p$ on $d$.

Following Hansen et al. [37], the second recursion theorem does not provide an effective solution for recursive programs. This observation changes if the programming language is reflexive. Indeed, it is more efficient to have a self-description than to have a complete copy of itself, which will be duplicated all along a computation. However, it is not the same thing when we are dealing with computer viruses. In this case, we really need to create an independent copy to contaminate a system. For this reason, and following Case [14], a virus needs a complete self-knowledge. This means that a virus may be able to generate an operational and independent copy of itself.

The construction of the fixpoint compiling function $\text{fix}$ is made, step-by-step, by defining four different program transformations. The parts mentioned below describe, in detail, each program transformation. The idea behind the construction is very similar to the von Neumann [13] method to build self-reproducing systems. This section concludes by stating both Kleene’s recursion theorem and Myhill’s variant.

(a) Write

We begin with an amazing hack of Moss [17]. A writing function $\text{wrt} : \{1, \#\}^* \rightarrow \{1, \#\}^*$ is a program transformation such that for every $p \in \{1, \#\}^*$,

$$\text{wrt}(p), d \Rightarrow \text{wrt}(p), d \bullet p. \quad (4.1)$$

In other words, $\text{wrt}(p)$ appends a copy of $p$ to the register $R_1$. We devise a program $\text{write}$, which computes $\text{wrt}(p)$ by generating a program that reconstructs its parameter $p$,

$$\text{write}, p, d_1, \ldots, d_n \Rightarrow \text{write}, \text{wrt}(p), d_1, \ldots, d_n. \quad (4.2)$$

We implement $\text{write}$ using $R_{n+2}$ as a temporary register,

```
1 move R_1, R_{n+2} (* move p *)
2 case n + 2
3 jmp 11 (* if p is empty *)
4 jmp 5 (* if p begins with # *)
5 put 1,1 (* otherwise p begins with 1 *)
6 put 1,1 (* insert the opcode [put 1,1] *)
7 put 1,# (* i.e. 11#, in R_1 *)
8 jmp −6 (* jump back to case instruction *)
```
From Turing machines to computer viruses

9 put 1,1 (* otherwise p begins with 1 *)
10 put 1,1 (* insert the opcode 'put 1,# 
11 put 1,# (* i.e. 11##, in R1. *)
12 put 1,#
13 jmp −11 (* jump back to case instruction *).

It is worth noticing that the write program is similar to quine constructions or to the Thompson [32] construction of a C self-reproducing program.

(b) Controller

The next program transformation is a function ctrl, whose task is to control the duplication as follows:

\[
ctrl(p), d_1, \ldots, d_n \Rightarrow ctrl(p), p, d_1, \ldots, d_n. \tag{4.3}
\]

The function ctrl(p) shifts registers to the right and then it drops a copy of p in the right register. This is performed by setting \( ctrl(p) = shift_n \bullet wrt(p) \). We check that equation (4.3) is satisfied,

\[
ctrl(p), d_1, \ldots, d_n \Rightarrow shift_n \bullet wrt(p), d_1, \ldots, d_n \\
\Rightarrow shift_n \bullet wrt(p), \epsilon, d_1, \ldots, d_n \\
\Rightarrow shift_n \bullet wrt(p), p, d_1, \ldots, d_n.
\]

We implement ctrl by the program ctrl defined as follows:

\[
ctrl(p), d_1, \ldots, d_n \Rightarrow ctrl, wrt(p), d_1, \ldots, d_n
\]

\[
write \Rightarrow ctrl, wrt(p), d_1, \ldots, d_n
\]

\[
move R_1, R_{n+2} \Rightarrow ctrl, \epsilon, d_1, \ldots, d_n, wrt(p)
\]

\[
wrt(shift_n) \Rightarrow ctrl, shift_n, d_1, \ldots, d_n, wrt(p)
\]

\[
move R_{n+2}, R_1 \Rightarrow ctrl, shift_n \bullet wrt(p), d_1, \ldots, d_n.
\]

From now on, we put on the right the sequential semantics of the program or of the instruction executed to help the reader. On the very first line, we indicate the initial configuration. The configuration on line 1 is obtained after executing the instruction write, and so on.

It follows that

\[
ctrl, p, d_1, \ldots, d_n \Rightarrow ctrl, ctrl(p), d_1, \ldots, d_n. \tag{4.4}
\]

Following §2e, the function ctrl inserts p correctly if we want to run it on \( d_1, \ldots, d_n \). We define ctrlX, which, in addition, runs p. We set \( ctrlX(p) = ctrl(p) \bullet exec 1 \), which thus satisfies

\[
ctrlX(p), d \Rightarrow ctrlX(p), p, d_1, \ldots, d_n \tag{4.5}
\]

\[
ctrlX(p), d \Rightarrow p, d_1, \ldots, d_n \tag{4.6}
\]

by (2.1).
In the first step, the program $\text{ctrl} X(p)$ shifts registers to the right and copies $p$. The second step consists of activating and running $p$ on $d$. As a result, the computation is identical to the computation of $(p, d_1, \ldots, d_n)$ because of the fact that a program is relocatable by a right translation.

Finally, note that $\text{ctrl} X(p)$ corresponds to $[\text{spec}](\text{interp}, p)$, or more precisely to $[\text{spec}](\text{exec 1}, p)$ in the SRM model. Actually, we can now easily devise a specialization function by setting $[\text{spec}](p, d) = \text{ctrl}(d) \cdot \text{ctrl} X(p)$ and thus,

$$[\text{spec}](p, d), d_1, \ldots, d_n \Rightarrow \text{ctrl}(d) \cdot \text{ctrl} X(p), d_1, \ldots, d_n \Rightarrow \text{ctrl}(d) \cdot \text{ctrl} X(p), d, d_1, \ldots, d_n \quad \text{by (4.3)}$$

$$\Rightarrow p, p, d, d_1, \ldots, d_n \quad \text{by (4.6)}.
$$

\hspace{1cm} (c) Diagonalizer

A function $\text{diag}: \{1, \#\}^* \rightarrow \{1, \#\}^*$ is a diagonalization function,

$$\text{diag}(p), d_1, \ldots, d_n \Rightarrow p, p, d_1, \ldots, d_n. \quad (4.7)$$

A diagonalizer $\text{diag}$ is a program that computes a diagonalization function,

$$\text{diag}, p, d_1, \ldots, d_n \Rightarrow \text{diag}, \text{diag}(p), d_1, \ldots, d_n. \quad (4.8)$$

We implement $\text{diag}$ by

$$\text{diag}, p, d_1, \ldots, d_n \Rightarrow \text{diag}, \text{ctrl}(p), d_1, \ldots, d_n$$

$$1 \text{ ctrl} \Rightarrow \text{diag}, \text{ctrl}(p), d_1, \ldots, d_n$$

$$2 \text{ cp } R_1, R_{n+2} \Rightarrow \text{diag}, \text{ctrl}(p), d_1, \ldots, d_n, \text{ctrl}(p)$$

$$3 \text{ move } R_{n+2}, R_1 \Rightarrow \text{diag}, \text{ctrl}(p) \cdot \text{ctrl}(p), d_1, \ldots, d_n$$

$$4 \text{ wrt } (\text{exec } 1) \Rightarrow \text{diag}, \text{ctrl}(p) \cdot \text{ctrl}(p) \cdot \text{exec } 1, d_1, \ldots, d_n.$$

As a result, $\text{diag}(p) = \text{ctrl}(p) \cdot \text{ctrl} X(p)$. We check that equation (4.7) is satisfied,

$$\text{diag}(p), d_1, \ldots, d_n \Rightarrow \text{ctrl}(p) \cdot \text{ctrl} X(p), d_1, \ldots, d_n \quad \text{by definition}$$

$$\Rightarrow \text{ctrl}(p) \cdot \text{ctrl} X(p), p, d_1, \ldots, d_n \quad \text{by (4.3)}$$

$$\Rightarrow \text{ctrl}(p) \cdot \text{ctrl} X(p), p, p, d_1, \ldots, d_n \quad \text{by (4.5)}$$

$$\Rightarrow p, p, d_1, \ldots, d_n \quad \text{by (2.1)}.$$

Note that a diagonalizer is a particular case of a specializer. It specializes a program with a copy of it. That is, $\text{diag}(p) = [\text{spec}](p, p)$.

\hspace{1cm} (d) Fixpoint compiler

A function $\text{fix}: \{1, \#\}^* \rightarrow \{1, \#\}^*$ is a fixpoint function if and only if for every $p \in \{1, \#\}^*$,

$$\text{fix}(p), d_1, \ldots, d_n \Rightarrow p, \text{fix}(p), d_1, \ldots, d_n. \quad (4.9)$$

Phil. Trans. R. Soc. A (2012)
We define \( \text{fix}(p) = \text{diag}(\text{diag} \bullet \text{ctrlX}(p)) \). We can check that (4.9) holds, 
\[
\text{fix}(p), d_1, \ldots, d_n \Rightarrow \text{diag}(\text{diag} \bullet \text{ctrlX}(p)), d_1, \ldots, d_n \quad \text{by definition}
\]
\[
\Rightarrow \text{diag} \bullet \text{ctrlX}(p), \text{diag} \bullet \text{ctrlX}(p), d_1, \ldots, d_n \quad \text{by (4.7)}
\]
\[
\Rightarrow \text{diag} \bullet \text{ctrlX}(p), \text{diag}(\text{diag} \bullet \text{ctrlX}(p)), d_1, \ldots, d_n \quad \text{by (4.8)}
\]
\[
\Rightarrow p, \text{diag}(\text{diag} \bullet \text{ctrlX}(p)), d_1, \ldots, d_n. \quad \text{by (2.1)}
\]

(e) Recursion theorems

We now state and demonstrate the Kleene [3] second recursion theorem in the SRM model.

**Theorem 4.1 (Kleene).** For any \( n \) and any program \( p \in \{1, \#\}^* \), there is a program \( e \in \{1, \#\}^* \) such that, for all \( d_1, \ldots, d_n \) in \( \{1, \#\}^* \),
\[
e, d_1, \ldots, d_n \Rightarrow p, e, d_1, \ldots, d_n. \quad (4.10)
\]

**Proof.** Put \( e = \text{fix}(p) \).

The uniform version of Myhill [36] is a direct consequence of the above construction. It constructs a fixpoint compiler from a given specification.

**Theorem 4.2 (Myhill).** There is a program \( \text{fix} \in \{1, \#\}^* \), such that for any program \( p \in \{1, \#\}^* \), for all \( d_1, \ldots, d_n \) in \( \{1, \#\}^* \),
\[
\text{fix}, p \Rightarrow \text{fix}, \text{fix}(p) \quad (4.11)
\]
and
\[
\text{fix}(p), d_1, \ldots, d_n \Rightarrow p, \text{fix}(p), d_1, \ldots, d_n. \quad (4.12)
\]

**Proof.** A fixpoint compiler \( \text{fix} \) is implemented as follows:

1. \( \text{ctrl} \) \( \text{fix}, p \Rightarrow \text{fix}, \text{ctrl}(p) \)
2. \( \text{wrt}(	ext{exec} 1) \Rightarrow \text{fix}, \text{ctrlX}(p) \)
3. \( \text{move} R_1, R_2 \Rightarrow \text{fix}, \epsilon, \text{ctrlX}(p) \)
4. \( \text{wrt}(	ext{diag}) \Rightarrow \text{fix}, \text{diag}, \text{ctrlX}(p) \)
5. \( \text{move} R_2, R_1 \Rightarrow \text{fix}, \text{diag} \bullet \text{ctrlX}(p) \)
6. \( \text{diag} \Rightarrow \text{fix}, \text{diag} \bullet \text{ctrlX}(p) \).

The program \( \text{fix} \) implements the fixpoint compiling function \( \text{fix} \).

5. Computer viruses

(a) Blueprint viruses

In Bonfante et al. [11], we define a class of blueprint viruses that duplicate themselves without modifying their own programs. The virus copy is injected inside a register. If the register is not empty, the virus copy is injected into the register data/program. Otherwise, a stand-alone offspring is created. Contrary to the earlier-mentioned study, the virus copy may be activated in the SRM MOC.
Thus, the virus copy may be seen as an independent organism, owing to the program motion property of the SRM. The first example given in the introduction is a blueprint virus. Adleman [9] proposed a model of computer viruses based on recursion theory. Several classes of viruses are defined. In each case, when a virus duplicates, its copy is identical to its source. In fact, viruses studied by Adleman are blueprint viruses.

The aim of this section is to show how blueprint virus construction depends on the recursion theorem, and how to construct a blueprint virus compiler.

Given a program $p$, a *blueprint virus* is a program $v$ that satisfies the equation

$$v, d_1, \ldots, d_n \Rightarrow p, v, d_1, \ldots, d_n. \quad (5.1)$$

In this equation, we say that $p$ is a viral specification. It is a program that has at least one argument, $v$, which denotes a copy of itself. In turn, $v$ may be executed and so it may pursue the infection. Let us return to the example of the introduction ($\S 1\alpha(i)$) in order to understand what is going on. In the introductory example, $\text{Specification\_of\_VA}$ corresponds to a viral specification of a blueprint virus, which we could formalize in the SRM model as follows:

$$v \Rightarrow \text{Specification\_of\_VA}, v.$$

The solution of the earlier-mentioned fixed-point equation is the virus that we are seeking for.

So, the construction of a blueprint virus corresponds to solving the fixpoint equation (5.1). The Kleene recursion theorem 4.1 gives an immediate solution by setting $v = \text{fix}(p)$.

It is important to see that Myhill’s recursion theorem allows us to define a compiler of viruses, which is called a *virus distribution*. A virus distribution is a program that takes a viral specification and outputs a virus that realizes it. Thus, $\text{fix}$ is a virus distribution. Let us now give two illustrations of blueprint viruses.

(i) *An overwriting virus*

We begin by a virus that erases one by one the content of each register. The specification of an overwriting virus is

$$\text{owrt}, x, d_1, \ldots, d_n \Rightarrow \text{owrt}, \epsilon, x, d_2, \ldots, d_n. \quad (5.2)$$

The program $\text{owrt}$ is implemented by

- $\text{erase } R_2 \Rightarrow \text{owrt}, x, \epsilon, \ldots, d_n$
- $\text{move } R_1, R_2 \Rightarrow \text{owrt}, \epsilon, x, \ldots, d_n$
- $\text{exec } 2 \Rightarrow \text{owrt}, \epsilon, x, \ldots, d_n.$

The recursion theorem gives a solution of equation (5.1) where $v_{\text{owrt}} = \text{fix}(\text{owrt})$,

$$v_{\text{owrt}}, d_1, \ldots, d_n \Rightarrow \text{owrt}, v_{\text{owrt}}, d_1, \ldots, d_n \quad \text{by (4.1)}$$

$$\Rightarrow \text{owrt}, \epsilon, v_{\text{owrt}}, d_2, \ldots, d_n \quad \text{by (5.2)}.$$
The computation goes on because the offspring $v_{\text{ovrt}}$ is activated,

\[
\text{ovrt}, \epsilon, v_{\text{ovrt}}, d_2, \ldots, d_n \Rightarrow \text{ovrt}, \epsilon, \text{ovrt}, v_{\text{ovrt}}, d_2, \ldots, d_n
\]

\[
\Rightarrow \text{ovrt}, \epsilon, \text{ovrt}, \epsilon, v_{\text{ovrt}}, d_3, \ldots, d_n
\]

\[
\Rightarrow \ldots
\]

The computation is repeated, and the virus thus spreads and erases the content of each register.

(ii) An ecto-symbiote

The next example is called an \textit{ecto-symbiote} because it lives on the surface of its host. The construction of an ecto-symbiote follows the lines of the previous virus. We define $\text{ecto} = \text{move } R_1, R_2 \cdot \text{exec } 2$. Thus, the program $\text{ecto}$ verifies the following equation:

\[
\text{ecto}, x, d_1, \ldots, d_n \Rightarrow \text{ecto}, x, d_1 \cdot x, \ldots, d_n.
\]

The viral specification $\text{ecto}$ says the ecto-symbiote moves a copy of itself and appends it on the register just on its right. Then, it runs the register on its right. If the computation of $d_1$ activates $x$, for example, by just halting normally, then the virus offspring will be run.

Again, the construction of the virus corresponds to solving this fixpoint equation,

\[
v_{\text{ecto}}, d_1, \ldots, d_n \Rightarrow \text{ecto}, v_{\text{ecto}}, d_1, \ldots, d_n.
\]

A solution is the fixpoint $v_{\text{ecto}} = \text{fix}(\text{ecto})$. By composing (5.3) and (5.4), we have

\[
v_{\text{ecto}}, d_1, \ldots, d_n \Rightarrow \text{ecto}, v_{\text{ecto}}, d_1 \cdot v_{\text{ecto}}, \ldots, d_n.
\]

(b) Cohen’s model of viruses

In his PhD thesis, Cohen [7] provided a model of computer viruses, which was published by Cohen [38], based on one-tape TMs. Informally, a virus is a code that is able to copy itself and to evolve on a given TM $M$. To take into account virus evolutions, Cohen’s model does not consider a single virus but a set $V$, named a \textit{viral set}, denoting the set of viruses of the machine $M$. All evolutions of a virus $v$ are in $V$. As a result, the model is a pair $VS = (M, V)$, where $M$ is a TM and $V$ is a viral set that satisfies the following property: if $M$ reads a virus $v \in V$, then $M$ will write an evolution $v' \in V$ on its tape. This definition is obviously recursive and $v'$ will produce in turn a word of $V$. To sum up, the two main traits of Cohen’s model are (i) viruses are self-reproducing programs and (ii) viruses may evolve. The next two parts are devoted to the construction of simple mutating viruses, which encompass Cohen’s definition with respect to an SRM model.

(c) Self-reproduction and semantics-preserving mutation

We now turn to the question of self-reproduction mechanisms in which the offspring mutates. There are several forms of possible mutations. Following

\[Phil. Trans. R. Soc. A (2012)\]
J.-Y. Marion

Rogers [31], there is an infinite number of solutions to a given viral specification. So there is a sequence of viruses \( v_1, v_2, \ldots, v_n, \ldots \) such that

\[
 v_1, d_1, \ldots, d_n \Rightarrow p, v_i, d_1, \ldots, d_n. \tag{5.5}
\]

Such a sequence of viruses may be obtained by applying a program transformation that preserves the original semantics. That is, a program transformation \( \text{Obs} \) such that \( v_{i+1} \) behaves exactly like \( \text{Obs}(v_i) \). From a theoretical point of view, a padding function works well. In practice, code obfuscation techniques are a current mutation method. Thus, the program denotation is concealed to a human analyst and it allows us to bypass detection procedures. Although Barak et al. [39] established that there is no good obfuscation method, obfuscation methods are very common in practice. Giacobazzi et al. [40] have showed relationships with meta-programming and code generation. In the same spirit, we have yet to analyse the role of the padding lemma as a theoretical basis of code obfuscation.

(d) Virus mutation

Till now, we have an infinite sequence of viruses with the same behaviour. However, an offspring is still an exact copy of its father because there is no in-built mutation mechanism. In practice, this means that the generation of a new mutant should be made by hand. Now, a direct application of the recursion theorem allows us to generate a different copy at each generation. For this, we need to solve the following fixed-point equation, where \( p \) denotes a viral specification:

\[
 v, d_1, \ldots, d_n \Rightarrow p, \text{Obs}(v), d_1, \ldots, d_n. \tag{5.6}
\]

Because \( \text{Obs} \) is a semantic-preserving transformation, we have

\[
 \text{Obs}(v), d_1, \ldots, d_n \Rightarrow p, \text{Obs} (\text{Obs}(v)), d_1, \ldots, d_n. \tag{5.7}
\]

If \( \text{Obs} \) is injective, then each virus mutation will be different. Thus, at each generation, the virus is different and operational. Because SRMs have the ability to activate offsprings, the viral infection will spread by injecting mutating viruses. A solution is also given by the second recursion theorem by setting \( v = \text{fix}(\text{obs} \bullet p) \), where \( \text{obs} \) computes \( \text{Obs} \), that is \( \text{obs}, q \Rightarrow \text{obs}, \text{Obs}(q) \).

6. Conclusion

We have considered only a small portion of the interesting fixpoint theorems. We have seen that a program may mutate by applying a semantics-preserving transformation. Other more elaborate recursion theorems give more complex viruses. Indeed, a program may also evolve by applying a non-semantic-preserving transformation, say \( \text{Evo} \). In this case, \( v \) and \( \text{Evo}(v) \) do not behave in the same way. We may think that from one generation to another, viruses evolve. In practice, this happens when a virus is updated either to apply a patch or to embed new functionalities. The construction of evolving viruses is not a direct consequence of the second recursion theorem. Indeed, we have to ensure that the evolution \( \text{Evo}(v) \) of \( v \) is able to reproduce itself. As a result, the construction of an evolving virus is based on stronger variants of the recursion theorem. A variant of the extended
fixpoint theorem of Smullyan [5] is well suited. In SRM settings, we can state it as follows:

\[ V(x), d_1, \ldots, d_n \Rightarrow x, V(Evo(x)), d_1, \ldots, d_n. \]

In this equation, \( V(x) \) is the evolving virus and \( x \) is its viral specification. The viral specification \( x \) is modified at each replication, owing to the \( Evo \) function. As a result, \( V(Evo(x)) \) is an operational virus, whose behaviour has been updated to compute \( Evo(x) \). The evolution consists of adding computational capabilities to self-replicating structures. There may be an analogy to explore with the works of Myhill [41] and an efficient construction of a conservative, and consistent hierarchy of arithmetic theories. Indeed, a similar construction would allow us to build viruses getting ‘smarter’. It is fascinating to see that most of the constructions in computability theory are now foundational ideas in computer science. To end this discussion, we may have a second look at the function \( V \) defined earlier. It is a program transformation that corresponds to a compiling function, but where the target program has a different behaviour to the source. The target program has evolved compared with the source.

An SRM MOC may also be seen as a simple abstract machine to study self-modifying programs. When an offspring is activated, self-reproduction is a particular case of self-modification. Actually, self-modifying programs are current. These are mostly used to speed up execution by using just-in-time-compilers or they are used to protect codes against human or automatic analysis. A self-modifying program generates at runtime its code. Thus, we can represent a self-modifying program by a sequence of code waves (i.e. code layers). Each wave is a non-self-modifying program, which is generated by a previous wave. The full program is formed by a set, or rather by a directed graph, of waves.

I thank Serge Grigorieff and Maurice Margenstern who have introduced me to computability theory and many other topics. I also thank Neil Jones for always profound talks and John Case, with whom I discussed this topic during his stay in Nancy.

References


