Immune Systems in Computer Virology

Guillaume Bonfante[1,2(✉)], Mohamed El-Aqqad[2], Benjamin Greenbaum[3,4], and Mathieu Hoyrup[1]

[1] Loria, B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France
Guillaume.Bonfante@loria.fr, Mathieu.Hoyrup@inria.fr
[2] École Nationale Supérieure des Mines de Nancy, Université de Lorraine, Nancy, France
Mohamed.El-aqqad1@etu.univ-lorraine.fr
[3] Icahn School of Medicine at Mount Sinai, New York, NY 10029, USA
Benjamin.Greenbaum@mssm.edu
[4] Institute for Advanced Study, Einstein Drive, Princeton, NJ 08540, USA

The analogy between computer viruses and biological viruses, from which computer viruses get their name [7], has been clear for the past several decades. During that time there has been progress in both understanding the vast diversity of biological viruses, and in abstract approaches to understanding computer viruses. However, there has not been a great deal of effort to see if the formal efforts in theoretical computer science can be of any use to our understanding of biological viruses.

In this work, we use biological viruses as a motivation to extend some well-known results in theoretical computers viruses. In Cohen's [7], a virus is a string which–read by a Turing Machine–reproduces either itself or a variant form of itself. In Adelman's [1] initial formalism, the theory of computer viruses was placed in the theory of recursive functions. One well known result from both theories is that the general problem of viral detection is undecidable, implying that general computer immune systems based on viral detection can always be circumvented and there are no robust ways to modify a detector successfully [5].

In biological systems, there is some notion that a biological virus is less powerful, computationally, than the host it infects. Motivated by that analogy, here we show two cases where viruses, due to diminished computational capacity relative to their hosts, will not always win. But, first, what is a virus? We state after Adelman [1] and Bonfante, Kaczmarek, Marion [3] that a virus $\mathbf{v}$ is a fix point in the sense of Kleene's Second Recursion Theorem:

$$[\![\mathbf{v}]\!](x) = f(\underline{\mathbf{v}}, x)$$

where f is called the propagation function which defines a viruses behavior in regard to its first argument. As justified in [3] or by Case and Moelius in [6], the model is strong enough to capture the virus mutability or even virus "factories". It is shown that the different versions of the Recursion Theorem–weak, strong, extended, double, see Smullyan's [12] for a precise terminology–correspond to different aspects of computer viruses.

G. Bonfante—The first author received the support of ANR-12-INSE-002.

A. Beckmann et al. (Eds.): CiE 2015, LNCS 9136, pp. 127–136, 2015.
DOI: 10.1007/978-3-319-20028-6_13

Fixed points exist as long as the framework is universal [11], that is Turing complete, with a universal function and a specializer. Thus the following defense strategy: to block viruses, one may simply prevent the existence of fixed points. As we will see, the Recursion Theorem holds as long as there is a specializer, projections and composition. It is hard to avoid the two latter criteria. Thus, we focus on systems without specializers. We provide a solution based on cons-free programs as it has been developed in the past by Jones [9]. Whatever the choice of enumeration of programs, there is no specializer for cons-free programs. It is worth noticing that the language is LOGSPACE-complete making it relatively powerful computationally speaking.

We develop an other scenario, perhaps closer to the analogy with biology. Our idea is to strengthen the defense against viruses, not to avoid their existence. Indeed, biological viruses exist, but their hosts have some defense capabilities. In this scenario, we suppose that there is a (finite) set of known viruses. Then, each time a program enters the system, it is submitted to a program (an immune cell) which verifies whether it behaves like one of the viruses and remove it accordingly. Our scenario is close to the strategy of anti-virus software: they (are supposed to) recognize infected programs relative to a malware database which contains the (finite) set of known viruses. The existence of such a detector infringes Rice's Theorem. Indeed, it corresponds to the decidability of program equivalence. Thus, and again, to get such a language, we will loose Turing completeness.

To our knowledge, in computer virology, only "negative" results have been established so far. They state more or less that there is no defenses against viruses. On the theoretical side, we refer the reader to the aforementioned work [1,7] which where followed by Zuo and Zhou [13] and then by Case and Moelius [5]. On a more practical side, there are also many interesting approaches. For instance, Borello and Mé [4] showed how metamorphism can trick anti-virus software. Other escaping techniques involve encryption, self-reproduction and feints, see [8] for a full survey. This contribution is a first attempt to provide "positive" solutions.

1 Introduction

An *alphabet* is a finite set Σ of *letters*. Given an alphabet Σ, let $\mathbb{T}_\Sigma$ be the set of binary trees with leaves in Σ, that is the smallest set containing Σ and $(\mathtt{t}_1 \cdot \mathtt{t}_2)$ whenever $\mathtt{t}_1, \mathtt{t}_2 \in \mathbb{T}_\Sigma$. The two functions π_1 and π_2 are the projections: $\pi_i(\mathtt{t}_1 \cdot \mathtt{t}_2) = \mathtt{t}_i$ and $\pi_i(\mathtt{c}) = \mathtt{c}$ with $i \in \{1, 2\}$ and $\mathtt{c} \in \Sigma$. The size of a tree $\mathtt{t}$ is denoted $|\mathtt{t}|$ and is defined by $|\mathtt{c}| = 1$, $\mathtt{c} \in \Sigma$, and $|(\mathtt{t} \cdot \mathtt{u})| = |\mathtt{t}| + |\mathtt{u}| + 1$.

A word $a_1 \cdot a_2 \cdots a_k$ in Σ^* is encoded in $\mathbb{T}_{\Sigma \cup \{\mathtt{nil}\}}$ as $(a_1 \cdot (a_2 \cdot (\cdots (a_k \cdot \mathtt{nil}) \cdots)))$ where $\mathtt{nil}$ is an atom used as an end-marker. This relates computations over words to the ones over trees.

Definition 1. *Let $\trianglelefteq$ be the sub-tree relation on $\mathbb{T}_\Sigma$, that is the smallest order (reflexive-transitive relation) such that for all $\mathtt{t}, \mathtt{u} \in \mathbb{T}_\Sigma$:*

– $\mathtt{t} \trianglelefteq \mathtt{t}$,
– $\mathtt{t} \trianglelefteq (\mathtt{t} \cdot \mathtt{u})$,
– $\mathtt{t} \trianglelefteq (\mathtt{u} \cdot \mathtt{t})$.

The embedding relation on $\mathbb{T}_\Sigma$ *is defined to be the smallest order such that* $\mathtt{t} \trianglelefteq \mathtt{u} \Rightarrow \mathtt{t} \underline{\blacktriangleleft} \mathtt{u}$, *and closed by context:* $\mathtt{t} \underline{\blacktriangleleft} \mathtt{t}' \wedge \mathtt{u} \underline{\blacktriangleleft} \mathtt{u}' \Rightarrow (\mathtt{t} \cdot \mathtt{u}) \underline{\blacktriangleleft} (\mathtt{u} \cdot \mathtt{u}')$.

Observe that $\mathtt{nil} \trianglelefteq (\mathtt{nil} \cdot \mathtt{nil})$. The difference between the sub-tree relation and the embedding one is exemplified by $(\mathtt{nil} \cdot (\mathtt{nil} \cdot \mathtt{nil})) \underline{\blacktriangleleft} ((\mathtt{nil} \cdot \mathtt{nil}) \cdot (\mathtt{nil} \cdot \mathtt{nil}))$ but $(\mathtt{nil} \cdot (\mathtt{nil} \cdot \mathtt{nil})) \not\trianglelefteq ((\mathtt{nil} \cdot \mathtt{nil}) \cdot (\mathtt{nil} \cdot \mathtt{nil}))$.

Let $\triangleleft$ and $\blacktriangleleft$ denote respectively the strict order relative to $\trianglelefteq$ and $\underline{\blacktriangleleft}$. From the definition, first, it is clear that if $\mathtt{t} \blacktriangleleft \mathtt{u}$, then $\mathtt{u} \not\trianglelefteq \mathtt{t}$. And, second, if $|\mathtt{t}| > |\mathtt{u}|$, then $\mathtt{t} \not\trianglelefteq \mathtt{u}$.

We present (a slight variant of) `While`, a generic imperative language introduced by Jones [9]. We suppose a given alphabet Σ contains an atom `nil`. Moreover, we suppose given (a denumerable set of) variables $\mathtt{Var} \ni \mathtt{X}_0, \mathtt{X}_1, \ldots$. In the following, $\mathtt{X}, \mathtt{Y}$ serve as generic variables. The syntax of `While` is given by the following grammar:

$$\begin{array}{lll} \mathtt{Expressions} \ni \mathtt{E}, \mathtt{F} & ::= & \mathtt{X} \mid \mathtt{t} \mid \mathtt{cons\ E\ F} \mid \mathtt{hd\ E} \mid \mathtt{tl\ E} \mid =?\ \mathtt{E\ F} \\ \mathtt{Commands} \ni \mathtt{C}, \mathtt{D} & ::= & \mathtt{X} := \mathtt{E} \mid \mathtt{C\ ;\ D} \mid \mathtt{while\ E\ do\ C} \\ \mathtt{Programs} \ni P & ::= & \mathtt{read\ X}_1, \ldots, \mathtt{X}_n\mathtt{;\ C;\ write\ Y} \end{array}$$

where $\mathtt{t} \in \mathbb{T}_\Sigma$.

1.1 Semantics of `While`

A configuration, next called a store, is a function $\sigma : \mathtt{Var} \to \mathbb{T}_\Sigma$. The set of stores is denoted $\mathfrak{S}_\Sigma$, or shorter, $\mathfrak{S}$ when Σ is clear from the context. Given a configuration $\sigma \in \mathfrak{S}$ a variable $\mathtt{X}$ and $\mathtt{t} \in \mathbb{T}_\Sigma$, $\sigma[\mathtt{X} \mapsto \mathtt{t}]$ is the store equal to σ on all variables but $\mathtt{X}$ for which it is set equal to $\mathtt{t}$.

The semantics of an expression $\mathtt{E}$ applied on a configuration σ is denoted $[\![\mathtt{E}]\!]\sigma$ and defined by the equations:

$$\begin{array}{lll} [\![\mathtt{X}]\!]\sigma = \sigma(\mathtt{X}) & [\![\mathtt{hd\ E}]\!]\sigma = \pi_1([\![\mathtt{E}]\!]\sigma) & [\![\mathtt{cons\ E\ F}]\!]\sigma = ([\![\mathtt{E}]\!]\sigma \cdot [\![\mathtt{F}]\!]\sigma) \\ [\![\mathtt{t}]\!]\sigma = \mathtt{t} & [\![\mathtt{tl\ E}]\!]\sigma = \pi_2([\![\mathtt{E}]\!]\sigma) & [\![=?\ \mathtt{E\ F}]\!]\sigma = [\![\mathtt{E}]\!]\sigma \simeq [\![\mathtt{F}]\!]\sigma \end{array}$$

where for equality $\simeq$, `nil` serves as false and $(\mathtt{nil} \cdot \mathtt{nil})$ as true. Each command $\mathtt{C} \in \mathtt{Commands}$ updates the store, that is $[\![\mathtt{C}]\!] : \mathfrak{S} \to \mathfrak{S}$ which is defined recursively as follows:

$$\begin{array}{lll} [\![\mathtt{X\ :=\ E}]\!]\sigma = \sigma[\mathtt{X} \mapsto [\![\mathtt{E}]\!]\sigma] & & \\ [\![\mathtt{C;D}]\!]\sigma = [\![\mathtt{D}]\!]([\![\mathtt{C}]\!]\sigma) & & \\ [\![\mathtt{while\ E\ do\ C}]\!]\sigma = \sigma & & \text{if } [\![\mathtt{E}]\!]\sigma = \mathtt{nil} \\ [\![\mathtt{while\ E\ do\ C}]\!]\sigma = [\![\mathtt{C\ ;while\ E\ do\ C}]\!]\sigma & & \text{otherwise} \end{array}$$

The program $\mathbf{p} \triangleq \mathtt{read\ X}_1, \ldots, \mathtt{X}_n\ \mathtt{C\ ;writeY}$ computes the following function. Given $\mathtt{t}_1, \ldots, \mathtt{t}_n$, in the initial configuration $\sigma_0(\mathtt{t}_1, \ldots, \mathtt{t}_n)$, all variables are set to `nil`, except $\mathtt{X}_1, \ldots, \mathtt{X}_n$ which are respectively set to $\mathtt{t}_1, \ldots, \mathtt{t}_n$. Then $[\![\mathbf{p}]\!](\mathtt{t}_1, \ldots, \mathtt{t}_n)$ is defined to be $([\![\mathtt{C}]\!]\sigma_0(\mathtt{t}_1, \ldots, \mathtt{t}_n))(\mathtt{Y})$.

1.2 While as an Acceptable Language

Let $\{\mathbf{assign}, \mathbf{seq}, \mathbf{while}, \mathtt{Var}, \mathbf{quote}, \mathbf{cons}, \mathbf{hd}, \mathbf{tl}, \mathbf{iseq}, \mathtt{nil}\}$ denote 10 distinct elements of $\mathbb{T}_\Sigma$. The representation $\underline{\mathbf{p}}$ of a program in While is defined recursively:

$$\begin{aligned}
\underline{0} &= \mathtt{nil} & \underline{\mathtt{X}_i} &= (\mathtt{Var} \cdot \underline{i}) \\
\underline{n+1} &= (\mathtt{nil} \cdot \underline{n}) & \underline{\mathtt{t}} &= (\mathbf{quote} \cdot t) \\
\underline{()} &= \mathtt{nil} & \underline{\mathtt{hd\ E}} &= (\mathbf{hd} \cdot \underline{\mathtt{E}}) \\
\underline{(x_1, \ldots)} &= (\underline{x_1} \cdot \underline{(\ldots)}) & \underline{\mathtt{tl\ E}} &= (\mathbf{tl} \cdot \underline{\mathtt{E}}) \\
\underline{\mathtt{X}_i := \mathtt{E}} &= (\mathbf{assign} \cdot (\underline{\mathtt{X}_i} \cdot \underline{\mathtt{E}})) & \underline{\mathtt{cons\ E\ F}} &= (\mathbf{cons} \cdot (\underline{\mathtt{E}} \cdot \underline{\mathtt{F}})) \\
\underline{\mathtt{C;D}} &= (\mathbf{seq} \cdot (\underline{\mathtt{C}} \cdot \underline{\mathtt{D}})) & \underline{=?\ \mathtt{E\ F}} &= (\mathbf{iseq} \cdot (\underline{\mathtt{E}} \cdot \underline{\mathtt{F}})) \\
\underline{\mathtt{while\ E\ do\ C}} &= (\mathbf{while} \cdot (\underline{\mathtt{E}} \cdot \underline{\mathtt{C}}))
\end{aligned}$$

and for a program, we define $\underline{\mathtt{read\ X}_1, \ldots, \mathtt{X}_n;\ \mathtt{C};\ \mathtt{write\ Y}} = (\underline{(\mathtt{X}_1, \ldots, \mathtt{X}_n)} \cdot (\underline{\mathtt{C}} \cdot \underline{\mathtt{Y}}))$.

More generally speaking, the representation of a programming language is an injective function from the set of programs (here While) to its corresponding data set (here $\mathbb{T}_\Sigma$).

As shown by Jones [9], there is a universal program $\mathbf{u} \in \mathtt{While}$, that is a program $\mathbf{u}$ such that for any program $\mathbf{p} \in \mathtt{While}$ and any data $\mathtt{t} \in \mathbb{T}_\Sigma$: $[\![\mathbf{u}]\!](\underline{\mathbf{p}}, \mathtt{t}) = [\![\mathbf{p}]\!](\mathtt{t})$. For all $m, n \in \mathbb{N}$, there is a specializer $\mathbf{s\_m\_n}$, that is a program $\mathbf{s\_m\_n}$ such that for all $m+n$-ary program $\mathbf{p}$, for all $\mathtt{t}_1, \ldots, \mathtt{t}_{m+n} \in \mathbb{T}_\Sigma$, $[\![[\![\mathbf{s\_m\_n}]\!](\underline{\mathbf{p}}, \mathtt{t}_1, \ldots, \mathtt{t}_m)]\!](\mathtt{t}_{m+1}, \ldots, \mathtt{t}_{m+n}) = [\![\mathbf{p}]\!](\mathtt{t}_1, \ldots, \mathtt{t}_{m+n})$. Finally, it is Turing-complete. Such a language is said to be acceptable in Jones/Roger's terms. As such, it is isomorphic to any other acceptable language as shown by Rogers:

Theorem 1 (Rogers *[11]*). *Two acceptable languages are isomorphic.*

That is there is a bijective computable function transforming programs in the first language to programs in the second one with equivalent semantics.

For any acceptable language, Kleene's second recursion theorem is known to hold. We recall:

Theorem 2 (Kleene's Second Recursion Theorem). *For any $k+1$-ary program $\mathbf{p}$, there is a k-ary program $\mathbf{e}$ satisfying for all inputs $\mathtt{t}_1, \ldots, \mathtt{t}_k \in \mathbb{T}_\Sigma$: $[\![\mathbf{e}]\!](\mathtt{t}_1, \ldots, \mathtt{t}_k) = [\![\mathbf{p}]\!](\underline{\mathbf{e}}, \mathtt{t}_1, \ldots, \mathtt{t}_k)$.*

Proof. For later use, we give a proof for $k = 1$. The proof for $k > 1$ follows the same schema. For the specializer $\mathbf{s\_1\_1} \triangleq \mathtt{read\ X}_0, \mathtt{X}_1; \mathtt{C}_{\mathbf{s\_1\_1}}; \mathtt{write\ Y}$, for all binary program $\mathbf{p} \in \mathcal{P}$ and $\mathtt{t}, \mathtt{t}' \in \mathbb{T}_\Sigma$, $[\![[\![\mathbf{s\_1\_1}]\!](\underline{\mathbf{p}}, \mathtt{t})]\!](\mathtt{t}') = [\![\mathbf{p}]\!](\mathtt{t}, \mathtt{t}')$. Let $\mathbf{p} = \mathtt{read\ X}'_0, \mathtt{X}'_1; \mathtt{C}_{\mathbf{p}}; \mathtt{write\ Y}'$. By renaming variables, we suppose without loss of generality that it does not share variables with $\mathbf{s\_1\_1}$. Then, let $\mathbf{r_p}$ be the program:

$$\begin{array}{l}
\mathtt{read\ X}''_0, \mathtt{X}''_1; \\
\mathtt{X}_0 := \mathtt{X}''_0;\ \mathtt{X}_1 := \mathtt{X}''_0; \\
\mathtt{C}_{\mathbf{s_1_1}}; \\
\mathtt{X}'_0 := \mathtt{Y};\ \mathtt{X}'_1 := \mathtt{X}''_1; \\
\mathtt{C}_{\mathbf{p}}; \\
\mathtt{write\ Y}'
\end{array}$$

with $\mathtt{X}_0'', \mathtt{X}_1''$ some fresh variables. Then, it is clear that for all $\mathbf{q} \in \mathcal{P}$ and all $\mathtt{t} \in \mathbb{T}_\Sigma$, $[\![\mathbf{r_p}]\!](\underline{\mathbf{q}}, \mathtt{t}) = [\![\mathbf{p}]\!]([\![\mathbf{s\_1\_1}]\!](\underline{\mathbf{q}}, \underline{\mathbf{q}}), \mathtt{t})$. Let $\mathbf{e} \triangleq \overline{[\![\mathbf{s\_1\_1}]\!](\underline{\mathbf{r_p}}, \underline{\mathbf{r_p}})}$, we get:

$$\begin{aligned}
[\![\mathbf{e}]\!](t) &= [\![[\![\mathbf{s_1_1}]\!](\underline{\mathbf{r_p}}, \underline{\mathbf{r_p}})]\!](\mathtt{t}) && \text{by def. of } [\![.]\!] \\
&= [\![\mathbf{r_p}]\!](\underline{\mathbf{r_p}}, \mathtt{t}) && \text{by def. of } \mathbf{s_1_1} \\
&= [\![\mathbf{p}]\!]([\![\mathbf{s_1_1}]\!](\underline{\mathbf{r_p}}, \underline{\mathbf{r_p}}), \mathtt{t}) && \text{by remark above} \\
&= [\![\mathbf{p}]\!](\underline{\mathbf{e}}, \mathtt{t}) && \text{since } \overline{[\![\mathbf{s_1_1}]\!](\underline{\mathbf{r_p}}, \underline{\mathbf{r_p}})} = [\![\mathbf{s_1_1}]\!](\underline{\mathbf{r_p}}, \underline{\mathbf{r_p}})
\end{aligned}$$

As justified by Bonfante, Kaczmarek and Marion in [3], a virus can be formalized as follows:

Definition 2 (Computer Virus). *Given a computable function B called the propagation function, a virus is a program* $\mathbf{v}$ *such that* $[\![\mathbf{v}]\!](\mathtt{t}) = B(\underline{\mathbf{v}}, \mathtt{t})$ *for all* $\mathtt{t} \in \mathbb{T}_\Sigma$.

In other words, it is a fixed point for a propagation function. Thus, as shown in [3], the second recursion theorem of Kleene implies that for any propagation function there is a corresponding virus. In other words, the theorem provides a virus compiler, and there are no general ways to avoid them. In the remaining, we restrict `While` to get around computer viruses. We propose two strategies to that end. First, we delineate a programming language in which the Recursion Theorem does not hold. As shown by the proof of the Recursion Theorem, the existence of a specializer, of composition and projection is sufficient to prove the Theorem. Thus, we find a programming language without specializer.

The second strategy consists in finding a language for which fixed point exists, but viruses can be detected. By detection we mean program equivalence as justified by Adleman in [1].

2 On Cons-Free Programs

$\mathtt{While}_{\setminus\{\mathtt{cons}\}}$ is the language `while` restricted to expressions of the shape:

$$\mathtt{Expressions} \ni \mathtt{E}, \mathtt{F} ::= \mathtt{X} \mid \mathtt{t} \mid \mathtt{hd\ E} \mid \mathtt{tl\ E} \mid \mathtt{E} =? \mathtt{F}$$

Such programs where initially considered by Jones under a complexity perspective. He proved that they compute exactly LOGSPACE predicates. We show that the Second Recursion Theorem does not hold in $\mathtt{While}_{\setminus\{\mathtt{cons}\}}$.

In this section, when $\mathtt{t} \in \mathbb{T}_\Sigma$ and $S \subseteq \mathbb{T}_\Sigma$, the notation $\mathtt{t} \trianglelefteq S$ means $\exists \mathtt{t}' \in S : \mathtt{t} \trianglelefteq \mathtt{t}'$. When S, S' are two sets, the notation $S \trianglelefteq S'$ states for $\forall \mathtt{t} \in S, \exists \mathtt{t}' \in S' : \mathtt{t} \trianglelefteq \mathtt{t}'$. For a store σ, let $\mathrm{Rg}(\sigma) = \{\sigma(\mathtt{X}) \mid \mathtt{X} \in \mathtt{Var}\}$.

Definition 3. *Let* `E` *be an expression, we denote by* $c(\mathtt{E})$ *the set of all constants occurring in* `E`*; formally, by induction:* $c(\mathtt{X}) = \emptyset$, $c(\mathtt{t}) = \{\mathtt{t}\}$, $c(\mathtt{hd\ E}) = c(\mathtt{tl\ E}) = c(\mathtt{E})$ *and* $c(\mathtt{cons\ E\ F}) = c(\mathtt{E}=?\ \mathtt{F}) = c(\mathtt{E}) \cup c(\mathtt{F})$.

The definition is extended to commands: $c(\mathtt{X} := \mathtt{E}) = c(\mathtt{E})$, $c(\mathtt{C\ ;\ D}) = c(\mathtt{C}) \cup c(\mathtt{D})$, $c(\mathtt{while\ E\ do\ C}) = c(\mathtt{E}) \cup c(\mathtt{C})$ *and finally to programs by the equation* $c(\mathtt{read\ X}_1, \ldots, \mathtt{X}_n; \mathtt{C};\ \mathtt{writeY}) = c(\mathtt{C})$.

Proposition 1. *Given a program* $\mathbf{p} \in \mathtt{While}_{\setminus\{\mathtt{cons}\}}$ *of arity* n *and* $\mathtt{t}_1, \cdots, \mathtt{t}_n$ *some elements of* $\mathbb{T}_\Sigma$, $[\![\mathbf{p}]\!](\mathtt{t}_1, \cdots, \mathtt{t}_n) \trianglelefteq c(\mathbf{p}) \cup \{\mathtt{t}_1, \cdots, \mathtt{t}_n\} \cup \{(\mathtt{nil} \cdot \mathtt{nil})\}$ *whenever* $[\![\mathbf{p}]\!](\mathtt{t}_1, \ldots, \mathtt{t}_n)$ *is defined.*

Proof. A very similar result occurs in Jones [9]. It is by induction on the structure of programs.

Proposition 2. *Given a program* $\mathbf{p} \in \mathtt{While}_{\setminus\{\mathtt{cons}\}}$ *and* $t \in \mathbb{T}_\Sigma$ *if* $\mathbf{p}$ *computes the constant function equal to* $\mathtt{t}$*, then either:*

$$\mathtt{t} \trianglelefteq (\mathtt{nil} \cdot \mathtt{nil}) \quad \textit{or} \quad \mathtt{t} \trianglelefteq c(\mathbf{p})$$

Proof. Applying Proposition 1 to the program $\mathbf{p}$: $[\![\mathbf{p}]\!](\mathtt{nil}) \trianglelefteq c(\mathbf{p}) \cup \{\mathtt{nil}, (\mathtt{nil} \cdot \mathtt{nil})\}$. But, again, $\mathtt{nil} \trianglelefteq (\mathtt{nil} \cdot \mathtt{nil})$, thus $[\![\mathbf{p}]\!](\mathtt{nil}) \trianglelefteq c(\mathbf{p}) \cup \{(\mathtt{nil} \cdot \mathtt{nil})\}$.

2.1 $\mathtt{While}_{\setminus\{\mathtt{cons}\}}$ Does Not Contain a Specializer

Theorem 3. *Whatever the choice of a representation, and in particular for the one given in the preceding section, there is no specializer in* $\mathtt{While}_{\setminus\{\mathtt{cons}\}}$.

Proof. We assume the existence of a specializer $\mathbf{s\_1\_1}$. Let us define two programs $\mathbf{p}$ and $\mathbf{q}$: $\mathbf{p} \triangleq \mathtt{read\ X_1, X_2;\ Y = X_1;\ write\ Y}$ and $\mathbf{q} \triangleq \mathtt{read\ X_1, X_2;\ if\ (X_2 = ?\ nil)\ Y := X_1\ else\ Y := X_2;\ write\ Y}$.

Consider some $\mathtt{t} \in \mathbb{T}_\Sigma$, we apply Proposition 1 to the program $\mathbf{s\_1\_1}$ and the inputs $\underline{\mathbf{p}}, \mathtt{t}$ we obtain:

$$\begin{cases} [\![\mathbf{s\_1\_1}]\!](\underline{\mathbf{p}}, \mathtt{t}) \trianglelefteq C \textit{ or} \\ [\![\mathbf{s\_1\_1}]\!](\underline{\mathbf{p}}, \mathtt{t}) \trianglelefteq \mathtt{t} \end{cases}$$

with $C = ((\mathtt{nil} \cdot \mathtt{nil}) \cdot \underline{\mathbf{p}} \cdot c_1 \cdots c_m)$ and $c(\mathbf{s\_1\_1}) = \{c_1, \cdots, c_m\}$.

Given $\mathtt{t} \neq \mathtt{t}'$, $\lceil [\![\mathbf{s\_1\_1}]\!](\mathbf{p}, \mathtt{t}) \rceil (\mathtt{nil}) = \mathtt{t} \neq \mathtt{t}' = \lceil [\![\mathbf{s\_1\_1}]\!](\mathbf{p}, \mathtt{t}') \rceil (\mathtt{nil})$. Thus, $\mathtt{t} \mapsto [\![\mathbf{s\_1\_1}]\!](\mathbf{p}, \mathtt{t})$ is an injective function. From that, we state that the set $S_C = \{\mathtt{t} \in \mathbb{T}_\Sigma \mid |[\![\mathbf{s\_1\_1}]\!](\mathbf{p}, \mathtt{t})| \leq |C|\}$ is finite. We set $N_1 = \max\{|\mathtt{t}| \mid \mathtt{t} \in S_C\}$. Then, for all $|\mathtt{t}| > N_1$, we can state that $|[\![\mathbf{s\_1\_1}]\!](\mathbf{p}, \mathtt{t})| > |C|$ which in turn means that $[\![\mathbf{s\_1\_1}]\!](\underline{\mathbf{p}}, \mathtt{t}) \trianglelefteq \mathtt{t}$.

Since for any $\mathtt{t} \neq \mathtt{t}'$, $\lceil [\![\mathbf{s\_1\_1}]\!](\mathbf{q}, \mathtt{t}) \rceil (\mathtt{nil}) = \mathtt{t} \neq \mathtt{t}' = \lceil [\![\mathbf{s\_1\_1}]\!](\mathbf{q}, \mathtt{t}') \rceil (\mathtt{nil})$, the function $\mathtt{t} \mapsto [\![\mathbf{s\_1\_1}]\!](\mathbf{q}, \mathtt{t})$ is injective. Thus, we use the same approach: there exists an integer N_2 such that $|\mathtt{t}| > N_2$ implies $[\![\mathbf{s\_1\_1}]\!](\underline{\mathbf{q}}, \mathtt{t}) \trianglelefteq \mathtt{t}$. We set $N = \max(N_1, N_2)$ and we get :

$$|\mathtt{t}| > N \Rightarrow ([\![\mathbf{s\_1\_1}]\!](\underline{\mathbf{p}}, \mathtt{t}) \trianglelefteq \mathtt{t} \ \textit{ and } \ [\![\mathbf{s\_1\_1}]\!](\underline{\mathbf{q}}, \mathtt{t}) \trianglelefteq \mathtt{t})$$

Given $n \in \mathbb{N}$, wet define $n_{\downarrow \trianglelefteq} = \{\mathtt{t} \in \mathbb{T}_\Sigma \mid \mathtt{t} \trianglelefteq \underline{n}\}$ with the encoding of integers defined in the preceding section. Observe that we have the equality:

$$n_{\downarrow \trianglelefteq} = \{\underline{k} \mid 0 \leq k \leq n\}. \tag{1}$$

For all $i, j \in \mathbb{N}$, let k such that $k \neq i$ and $k \neq j$. We have $[\![[\![\mathbf{s\_1\_1}]\!](\underline{\mathbf{p}}, \underline{i})]\!](\underline{k}) = \underline{i} \neq \underline{k} = [\![[\![\mathbf{s\_1\_1}]\!](\underline{\mathbf{q}}, \underline{j})]\!](\underline{k})$ which means that for all $i, j \in \mathbb{N}$: $[\![\mathbf{s\_1\_1}]\!](\underline{\mathbf{p}}, \underline{i}) \neq [\![\mathbf{s\_1\_1}]\!](\underline{\mathbf{q}}, \underline{j})$.

Consider some $M > N$. Recall that both $\mathtt{t} \mapsto [\![\mathbf{s\_1\_1}]\!](\mathbf{p}, \mathtt{t})$ and $\mathtt{t} \mapsto [\![\mathbf{s\_1\_1}]\!](\mathbf{q}, \mathtt{t})$ are injective. Due to the aforementioned remark, the set $S_{MN} = \{[\![\mathbf{s\_1\_1}]\!](\mathbf{p}, \underline{i}) \mid N < i \leq M\} \cup \{[\![\mathbf{s\_1\_1}]\!](\mathbf{q}, \underline{i}) \mid N < i \leq M\}$ contains exactly $2 \times (M - N)$ elements. However, all these elements verify $S_{MN} \trianglelefteq \underline{M} \in M_{\downarrow \trianglelefteq}$, but by Eq. 1, the set $M_{\downarrow \trianglelefteq}$ contains only $M + 1$ elements which leads to $2 \times (M - N) \leq M + 1$. The inequality does not hold for $M = 2 \times (N + 1)$.

The non-existence of a specializer does not mean that there are no fixed points in $\mathtt{While}_{\setminus\{\mathtt{cons}\}}$, for instance the nowhere defined program

$$\mathtt{read\ X_1}\ ;\mathbf{while}\ (\mathtt{nil} \cdot \mathtt{nil})\ \mathtt{do\ X_1} :=\ \mathtt{X_1}\ ;\ \mathtt{write\ X_1}$$

is a fixed point for the program

$$\mathtt{read\ X_1, X_2}\ ;\mathbf{while}\ (\mathtt{nil} \cdot \mathtt{nil})\ \mathtt{do\ X_1}\ :=\ \mathtt{X_1}\ ;\mathtt{write\ X_1}.$$

There are other fix-point constructions which are not based on specializers, some are found in Smullyan's [12], but we wish to mention here the approach due to Moss [10] which is based on a very elementary framework, text register machines.

Nevertheless, there are programs for which there is no fixed points. In other words, the Recursion Theorem does not hold. For instance, for the homeomorphic representation of programs presented in $\mathtt{While}_{\setminus\{\mathtt{cons}\}}$:

Proposition 3. *There is no Quine in* $\mathtt{While}_{\setminus\{\mathtt{cons}\}}$.

Proof. Ad absurdum, suppose there is a Quine $\mathbf{q}$ in $\mathtt{While}_{\setminus\{\mathtt{cons}\}}$. It is a fixed point for the program $\mathbf{pi_1} \triangleq \mathtt{read\ X_1, X_2\ ; X_1\ :=\ X_1\ ; write\ X_1}$. Since $[\![\mathbf{q}]\!](\mathtt{nil}) = \underline{\mathbf{q}}$, since $|\underline{\mathbf{q}}| > |(\mathtt{nil} \cdot \mathtt{nil})|$ (as it is the case for any programs), we can state with Corollary 1 that $\underline{\mathbf{q}} \trianglelefteq c(\mathbf{q})$. Let $\mathtt{t} \in c(\mathbf{q})$ such that

$$\underline{\mathbf{q}} \trianglelefteq \mathtt{t}. \tag{2}$$

With the homeomorphic encoding we chose, we can state that $(\mathbf{quote} \cdot \mathtt{t}) \trianglelefteq \underline{\mathbf{q}}$. Thus

$$\mathtt{t} \triangleleft (\mathbf{quote} \cdot \mathtt{t}) \trianglelefteq \underline{\mathbf{q}} \tag{3}$$

The two inequalities 2 and 3 are not compatible. The conclusion follows.

Quines are interesting in Adleman's perspective. They correspond to the 'Imitate' scenario. The 'Infection' scenario would not be possible due to Proposition 1. Thus, programs in $\mathtt{While}_{\setminus\{\mathtt{cons}\}}$ cannot be infected in his view. The reader may notice that the proof here depends on the choice of the representation of programs. Indeed, it is not difficult to define an encoding for which there is a Quine. Simply modify $\underline{\ }$ so that the encoding of $\mathbf{nil} \triangleq \mathtt{read\ X_1\ ;\ X_1\ :=\ nil\ ;write\ X_1}$ is set to $\mathtt{nil}$. Then, $[\![\mathbf{nil}]\!](\mathtt{t}) = \mathtt{nil}$ which is the required equation. Nevertheless, there are no other Quines. To end the remark, observe that program representation can be on the defense side, not on the virus writer's one.

3 Tiny, a Whippersnapper Programming Language

Tiny is the language While restricted to expressions of the shape:

$$\texttt{Expressions} \ni \texttt{E}, \texttt{F} ::= \texttt{X} \mid \texttt{t} \mid \texttt{cons E F} \mid \texttt{hd E} \mid \texttt{tl E}$$

and commands to:

$$\texttt{Commands} \ni \texttt{C}, \texttt{D} ::= \texttt{X} := \texttt{E} \mid \texttt{C ; D}.$$

Obviously, Tiny is not Turing complete. Actually, it is a very weak fragment of computable functions: it contains only functions computable in constant time. However, the Recursion Theorem holds, surprisingly, in Tiny. For the representation of programs that we defined, the Recursion Theorem holds:

Theorem 4. *Given a k-ary program $\mathbf{p} \in \texttt{Tiny}$, there is a $k-1$-ary program $\mathbf{e} \in \texttt{Tiny}$ such that for all $\texttt{t}_1, \ldots, \texttt{t}_k \in \mathbb{T}_\Sigma$: $[\![\mathbf{e}]\!](\texttt{t}_1, \ldots, \texttt{t}_k) = [\![\mathbf{p}]\!](\underline{\mathbf{e}}, \texttt{t}_1, \ldots, \texttt{t}_k)$.*

Proof. Again, we give a proof for $k = 1$. The other cases are left to the reader. If we come back to the previous proof of the Recursion Theorem, it is clear that the program $\mathbf{r_p}$ is in Tiny whenever both $\mathbf{p}$ and **s_1_1** are in Tiny. Since $\mathbf{p} \in \texttt{Tiny}$ by hypothesis, $\mathbf{r_p}$ is in Tiny if there is a specializer within Tiny. This is actually the case: define **s_1_1** $\triangleq$

```
read X0, X1;
C := hd (tl X0);              // the representation of the body of X0
X := hd (hd X0);              // the rep. of the first input variable of X0
XL := tl (hd X0);             // the remaining variables of X0
Y := tl (tl X0);              // the rep. of the output variable of X0
E := cons quote X1;           // the rep. of the value t of X1
C0 := cons assign (cons X E); // the rep of X := t
C := cons seq (cons C0 C);    // the rep. of X := t ; C
P := cons XL(cons C Y);       // the packaging of the new (unary) program
write P
```

This is a specializer. Indeed, let $\mathbf{p} = \texttt{read X}'_0, \texttt{X}'_1; \texttt{C}_\mathbf{p}; \texttt{write Y}'$. For all $\texttt{t} \in \mathbb{T}_\Sigma$, $[\![\mathbf{s\_1\_1}]\!](\underline{\mathbf{p}}, \texttt{t}) = \overline{\texttt{read X}'_1; \texttt{X}'_0 := \texttt{t} \,; \texttt{C}_\mathbf{p}; \texttt{write Y}'}$. Thus, for all $\texttt{t}' \in \mathbb{T}_\Sigma$: we have $[\![[\![\mathbf{s\_m\_n}]\!](\underline{\mathbf{p}}, \texttt{t})]\!]\overline{(\texttt{t}')} = [\![\mathbf{p}]\!](\texttt{t}, \texttt{t}')$ as required.

So, $\mathbf{r_p}$ is in Tiny. Since the fixed point $\mathbf{e} = \overline{\texttt{S}^1_1(\mathbf{r_p}, \mathbf{r_p})}$ is in Tiny, the proof ends as a corollary of the following Lemma:

Lemma 1. *If $\mathbf{p} \in \texttt{Tiny}$, for all $\texttt{t} \in \mathbb{T}_\Sigma$: $\overline{[\![\mathbf{s\_1\_1}]\!](\underline{\mathbf{p}}, \texttt{t})} \in \texttt{Tiny}$.*

Proof. Recall that $[\![\mathbf{s\_1\_1}]\!](\underline{\mathbf{p}}, \texttt{t}) = \overline{\texttt{read X}'_1; \texttt{X}'_0 := \texttt{t} \,; \texttt{C}_\mathbf{p}; \texttt{write Y}}$. Since $\mathbf{p}$ is in Tiny, $[\![\mathbf{s\_1\_1}]\!](\underline{\mathbf{p}}, \texttt{t})$ is itself the representation of a program in Tiny.

3.1 Program Equivalence in Tiny

From the above, there are viruses in Tiny. However, with the scenario made in the introduction, we can protect a system which is based on Tiny. Protection amounts to problem equivalence decision. It is the following. Given two programs $\mathbf{p}, \mathbf{q}$, does $[\![\mathbf{p}]\!] = [\![\mathbf{q}]\!]$? In general—in particular for a Turing-complete Language—, such a decision is not computable (as a direct consequence of Rice's Theorem). But, for Tiny, there is a simple decision procedure.

Theorem 5. *Equivalence of programs is computable for programs in* Tiny.

Proof. Composing expressions, one may reduce programs in Tiny to just one expression. The semantics of an expression can be explicitly expanded. Then, equivalence is equality of the semantics. We provide in appendix an algorithm that compute the semantics of expressions. However,

Proposition 4. *Equalence of programs in* Tiny *is not in* Tiny.

Lemma 2. *Any program in* Tiny *is monotonic, that is if* $\mathtt{t}_i \blacktriangleleft \mathtt{t}'_i$ *for all* $i \leq n$, *then* $[\![\mathbf{p}]\!](\mathtt{t}_1, \ldots, \mathtt{t}_n) \blacktriangleleft [\![\mathbf{p}]\!](\mathtt{t}'_1, \ldots, \mathtt{t}'_n)$.

Proof. We have seen above that any program in Tiny is equivalent to some program of the shape `read X`$_1, \ldots,$ `X`$_n$`; Y := E; write Y`, thus we restrict our attention to these ones. Since π_1 and π_2 are monotonic, the result holds by an immediate induction on `E`.

Proof (Proposition 4). Let us come back to the proof of the Proposition. Ad absurdum, suppose that there is some program $\mathbf{eq} \in$ Tiny such that $[\![\mathbf{eq}]\!](\underline{\mathbf{p}_1}, \underline{\mathbf{p}_2}) \neq \mathtt{nil}$ iff $[\![\mathbf{p}_1]\!] = [\![\mathbf{p}_2]\!]$ for all $\mathbf{p}_1, \mathbf{p}_2 \in$ Tiny. Let $\mathbf{p}_1 \triangleq$ `read X; Y := nil; write Y`. It is in tiny, thus, $[\![\mathbf{eq}]\!](\underline{\mathbf{p_1}}, \underline{\mathbf{p_1}}) \neq \mathtt{nil}$ since $\mathbf{p}_1$ is equivalent to itself. Observe that $\mathbf{p}_2 \triangleq$ `read X; Y := cons nil nil; write Y` which is also in Tiny verifies $\underline{\mathbf{p}_1} \blacktriangleleft \underline{\mathbf{p}_2}$. By Lemma 2, we can state that $[\![\mathbf{eq}]\!](\underline{\mathbf{p_1}}, \underline{\mathbf{p_1}}) \blacktriangleleft [\![\mathbf{eq}]\!](\underline{\mathbf{p_1}}, \underline{\mathbf{p_2}})$. In turn, that means $[\![\mathbf{eq}]\!](\underline{\mathbf{p_1}}, \underline{\mathbf{p_2}}) \neq \mathtt{nil}$. But $\mathbf{p}_1$ and $\mathbf{p}_2$ are not equivalent: $[\![\mathbf{p}_1]\!](\mathtt{nil}) \neq [\![\mathbf{p}_2]\!](\mathtt{nil})$, thus a contradiction.

4 Conclusion

Thought it is conceptually deep, the Recursion Theorem can be difficult to utilize for practical applications. In the context of computer viruses, it can often have a negative flavor. To our mind, our work opens a new branch of research which constructively studies fixed point within constrained computation systems. Types, logics and weak arithmetics arise as good candidates for that sake.

We end with a side remark about the efficiency of fixed points. Let us cite Hansen, Nikolajsen, Träff and Jones in [2]: "[...] running a fixed-point program to compute the factorial of n results in n levels of interpretation, each one slowing down execution by a large constant factor". This leads the authors to introduce a self-reflection statement that, supposedly, enables efficient fixed points.

The fixed points presented above never involve any interpretation layer. The construction of the specializer shows that it only introduce a constant time overhead with respect to the initial program. Therefore, the complexity of the fixed point **e** is equal to the one of the program $\mathbf{r_p}$. It involves the code of **s_1_1** which is in `Tiny`, and thus takes constant time and so few assignment which do not increase the size of their inputs. In the end, we see that the program **e** is as efficient as **p** on its input up to a constant factor.

Acknowledgements. The authors would like to thank the Institute for Advanced Study and the organizers of the 2012 Program in Theoretical Physics on Biology and Computation. In particular, we would like to thank Stanislas Leibler for several motivating discussions.

References

1. Adleman, L.M.: An abstract theory of computer viruses. In: Goldwasser, S. (ed.) CRYPTO 1988. LNCS, vol. 403, pp. 354–374. Springer, Heidelberg (1990)
2. Amtoft, T., Thomas, H., Jesper, N., Träff, L., Jones, N.D.: Experiments with implementations of two theoretical constructions. In: Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, PLAS 2007, San Diego, California, USA, 14 June 2007, pp. 47–52. ACM (2007)
3. Bonfante, G., Kaczmarek, M., Marion, J.-Y.: A classification of viruses through recursion theorems. In: Cooper, S.B., Löwe, B., Sorbi, A. (eds.) CiE 2007. LNCS, vol. 4497, pp. 73–82. Springer, Heidelberg (2007)
4. Borello, J.-M., Mé, L.: Code obfuscation techniques for metamorphic viruses. J. Comput. Virol. **4**(3), 211–220 (2008)
5. Case, J., Moelius, S.E.: Cautious virus detection in the extreme. In: Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, PLAS 2007, San Diego, California, USA, 14 June 2007, pp. 47–52. ACM (2007)
6. Case, J., Moelius, S.E.: Characterizing programming systems allowing program self-reference. Theory Comput. Syst. **45**(4), 756–772 (2009)
7. Cohen, F.: Computer viruses: theory and experiments. Comput. Secur. **6**(1), 22–35 (1987)
8. Collberg, C., Nagra, J.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection, 1st edn. Addison-Wesley Professional, New Jersey (2009)
9. Jones, N.D.: Computability and Complexity, from a Programming Perspective. MIT press, Cambridge (1997)
10. Moss, L.S.: Recursion theorems and self-replication via text register machine programs. Bull. EATCS **89**, 171–182 (2006)
11. Rogers Jr., H.: Theory of Recursive Functions and Effective Computability. McGraw Hill, New York (1967)
12. Smullyan, R.M.: Recursion Theory for Metamathematics. Oxford University Press, Oxford (1993)
13. Zuo, Z., Zhou, M.: Some further theorical results about computer viruses. Comput. J. **47**(6), 627–633 (2004)