A Classification of Viruses through Recursion Theorems

Guillaume Bonfante, Matthieu Kaczmarek, Jean-Yves Marion

A Classification of Viruses through Recursion Theorems

Guillaume Bonfante, Matthieu Kaczmarek and Jean-Yves Marion

Guillaume.Bonfante@loria.fr, Matthieu.Kaczmarek@loria.fr and Jean-Yves.Marion@loria.fr
Loria - INPL - Ecole Nationale Supérieure des Mines de Nancy
B.P. 239, 54506 Vandœuvre-lès-Nancy Cédex, France

Abstract. We study computer virology from an abstract point of view. Viruses and worms are self-replicating programs, whose definitions are based on Kleene's second recursion theorem. We introduce a notion of delayed recursion that we apply to both Kleene's second recursion theorem and Smullyan's double recursion theorem. This leads us to define four classes of viruses, two of them being polymorphic. Then, we work on a simple imperative programming language in order to show how those theoretical constructions can be implemented. In particular, we propose a general virus builder, and distribution engines.
Topics covered. Computability theoretic aspects of programs, computer virology.
Keywords. Computer viruses, polymorphism, propagation, recursion theorem, iteration theorem.

1 Theoretical Computer Virology

An important information security breach is computer virus infections. Following Filiol's book [10], we do think that theoretical studies should help to design new defenses against computer viruses. The objective of this paper is to pursue a theoretical study of computer viruses initiated in [4]. It is worth to cite von Neumann [26].

> Can an automaton be constructed, i.e., assembled and built from appropriately "raw material", by an other automaton? [...] Can the construction of automata by automata progress from simpler types to increasingly complicated types?

Since viruses are essentially self-replicating programs, we see that virus programming methods are an attempt to answer to von Neumann's question.

A long term practical motivation of this work is to suggest new research directions in order to improve anti-virus techniques. Of course, we have to keep in mind that Cohen demonstrated that finding a virus is undecidable in the general case. However, as the problem of defense still remains, various detection techniques have been developped. They are based on the combination of signature pattern matching and behavioural detection methods, see [11]. We think that a

first step to reach an immunization technique is to understand the key concepts of virus writing. According to Thompson [25] and Ludwig [17], this is not trivial.

Abstract computer virology was initiated in the 80's by the seminal works of Cohen and Adleman [7]. The latter coined the term *virus*. Cohen defined viruses with respect to Turing Machines [8]. Later [1], Adleman took a more abstract point of view in order to have a definition independent from any particular computational model. Then, only a few theoretical studies followed those seminal works. Chess and White refined the mutation model of Cohen in [6]. Zuo and Zhou formalized polymorphism from Adleman's work [27] and they analyzed the time complexity of viruses [28].

From [23, 19], it is commonly accepted that a programming language enjoys self-reference as long as it satisfies the recursion theorem, self-reference being the central issue in computer virology. Moreover, implementations using the recursion theorem are efficient as was shown by Jones [15] (up to a linear constant). Suppose that a programming environment provides a way to get its code to the currently executed program. For example, in `bash` this is given by the variable `$0`. This feature does not offer more efficient self-referential codes. Our constructions show moreover that viruses are not easier (at least theoretically) to write with than without built-in self-reference. That is why we focus on Kleene's recursion theorem, and consequently, we have chosen the core language $\texttt{WHILE}^+$ which fulfils the theorem.

Despite the works [12, 13], the recursion theorem is used essentially to prove "negative" results such as the constructions of undecidable or inseparable sets, see [22] for a general reference, or such as Blum's speed-up theorem [2]. Here, we show that the recursion theorem plays a key role in the *construction* of viruses.

Recently, we tried [3, 4] to formalize inside computability the notion of viruses. This formalization captures previous definitions that we have mentioned above. We also have characterized two kinds of viruses, blueprint and smith viruses, and we proved constructively their existence. The present work proposes to go further, introducing a notion of distribution to model polymorphism or metamorphism and a notion of virus distribution builder, a virus generator. Furthermore, we switch to a simple programming language named $\texttt{WHILE}^+$, introduced by Jones, to illustrate the effectivity of our constructions.

The concept of distribution is independent from any mutation technique and consequently a good model to study polymorphism or metamorphism. We have defined four kinds of viruses, the blueprint, the smith and their polymorphic variants. We show that each category is closely linked to a corresponding form of the recursion theorem. Kleene's recursion theorem corresponds to blueprint viruses, a model of duplication processes. A variation of Kleene's theorem introduces a code a of fixed point generators. This result provides a construction of polymorphic viruses based on a blueprint duplication. A second variation presents the double recursion theorem, which defines the class of smith viruses. This class of viruses models self-propagating infections. Lastly, the third variation is a delayed double recursion theorem, which leads to polymorphic smith viruses.

All our existential results are constructive. We give for each theoretical construction a code in $\texttt{WHILE}^+$. Actually, we follow the ideas of the experimentation of the iteration theorem and of the recursion theorem, which are developed in [12, 13] by Jones et al. and very recently by Moss in [18].

2 A Virus Definition

2.1 The $\texttt{WHILE}^+$ language

The domain of computation $\mathbb{D}$ is the set of binary trees generated from an atom `nil` and a (computable) pairing mechanism $\langle\ ,\ \rangle$. We suppose we are given a countable set $\mathbb{V}$ of variables and an other (countable) set of names of programs, say $\mathbb{N}$. The syntax of $\texttt{WHILE}^+$ is given by the following grammar.

$$\begin{aligned}
\text{Expressions: } \mathbb{E} \to\ & \mathbb{D} \mid \mathbb{V} \mid \texttt{cons}(\mathbb{E}_1, \mathbb{E}_2) \mid \texttt{hd}(\mathbb{E}) \mid \texttt{tl}(\mathbb{E}) \mid \\
& \texttt{exec}(\mathbb{E}_0, \mathbb{E}_1, \ldots, \mathbb{E}_n) \mid \texttt{spec}_n(\mathbb{E}_0, \mathbb{E}_1 \ldots, \mathbb{E}_n) \text{ with } n \geq 1 \\
\text{Commands: } \mathbb{C} \to\ & \mathbb{V} := \mathbb{E} \mid \mathbb{C}_1; \mathbb{C}_2 \mid \texttt{while}(\mathbb{E})\{\mathbb{C}\} \mid \texttt{if}(\mathbb{E})\{\mathbb{C}_1\}\texttt{else}\{\mathbb{C}_2\} \\
\text{Programs: } \mathbb{P} \to\ & \mathbb{N}(\mathbb{V}_1, \ldots, \mathbb{V}_n)\{\mathbb{C}; \texttt{return } \mathbb{E};\}
\end{aligned}$$

Elements of $\mathbb{P}$ are $\texttt{WHILE}^+$ programs. We suppose that we are given a concrete syntax of $\texttt{WHILE}^+$, that is an embedding of $\mathbb{P}$ into $\mathbb{D}$, see Appendix A. So, from now on, when the context is clear, we do not make any distinction between a program and its concrete syntax. Moreover, as a shorthand, to denote a program, we will use its name.

The semantics of $\texttt{WHILE}^+$ is the usual one and we postpone it to the Appendix B. $[\![\mathbf{p}]\!]$ denotes the function computed by the program $\mathbf{p}$. For convenience, we have chosen to have a built-in interpreter $\texttt{exec}(e, e_1, \ldots, e_n)$ which executes the evaluation of the expression e on the evaluation of $e_1, \ldots, e_n$. We also use a built-in specializer. Let us introduce the programs:

$\mathbf{spec_n}\ (x_0, \ldots, x_n)\ \{$
 r $:=$ $\texttt{spec}(x_0, \ldots, x_n)$;
 `return` r;
}

We have $[\![[\![\mathbf{spec_n}]\!](\mathbf{p}, x_1 \ldots, x_n)]\!](y) = [\![\mathbf{p}]\!](x_1, \ldots, x_n, y)$.

The use of an interpreter and of a specializer is justified by Jones who showed in [15] that programs with these constructions can be simulated up to a linear constant time by programs without them.

If f and g designate the same function, we write $f \approx g$. A function f is *semi-computable* if there is a program $\mathbf{p}$ such that $[\![\mathbf{p}]\!] \approx f$, moreover, if f is total, we say that f is *computable*.

2.2 What is a Computer Virus?

We do think that Kleene's recursion theorem is the mathematical backbone of computer virology. Several authors made the same observation [21, 19, 1, 10], but as far as we know, none have developed this idea into a framework.

To model viruses, we propose the following scenario. When a program $\mathbf{p}$ is executed within an environment x, we compute $[\![\mathbf{p}]\!](x)$ and if the evaluation halts, we replace x by the result of the computation. The entry x is thought of as a finite sequence $\langle x_1, \ldots, x_n \rangle$ which represents the file system and all accessible parameters.

Let us provide some examples to fix the intuition. Typically, a program **copy** which copies a file satisfies $[\![\mathbf{copy}]\!](\mathbf{p}, x) = \langle \mathbf{p}, \mathbf{p}, x \rangle$. The original environment is $\langle \mathbf{p}, x \rangle$. After the evaluation of **copy**, we have the environment $\langle \mathbf{p}, \mathbf{p}, x \rangle$. The particular element $\mathbf{p}$ has been copied.

Our second example models a *parasitic virus* [10]. Parasitic viruses insert themselves into existing files. When an infected host is executed, first the virus infects a new host, then it gives the control back to the original host. We write $x; y$ to denote the composition of x and y and **id** is a code of the identity. Let us define $B(y, \mathbf{p}) = [\![\mathbf{spec_1}]\!](y, \mathbf{id}); \mathbf{p}$. The program $B(y, \mathbf{p})$ is the "infected form" of $\mathbf{p}$ by y. Suppose that a program $\mathbf{v}$ satifies:

$$[\![\mathbf{v}]\!](\mathbf{p}, \mathbf{q}, x) = [\![\mathbf{p}]\!](B(\mathbf{v}, \mathbf{q}), x) \tag{1}$$

The following equations hold:

$$\begin{aligned} [\![B(\mathbf{v}, \mathbf{p})]\!](\mathbf{q}, x) &= [\![\mathbf{p}]\!](B(\mathbf{v}, \mathbf{q}), x) && \text{by specialization} && (2) \\ &= [\![\mathbf{v}]\!](\mathbf{p}, \mathbf{q}, x) && \text{by 1.} && (3) \end{aligned}$$

Equation (2) shows the propagation process of the scenario. Equation (3) expresses the fact that an infected form can be seen as a specialized form of the virus $\mathbf{v}$ for the host $\mathbf{p}$.

Generally speaking, the construction of viruses lies in the resolution of equations such as (1, 3) above with $\mathbf{v}$ and B as unknowns. The latter equation links the virus to its infected forms while the former specifies the way the infection is propagated. The infection, expressed here by Equation (2), is then a consequence of the two equations (1, 3). We will show how this resolution can be done by use of Kleene's recursion theorem and specialization.

With the previous discussion in mind, we present a formalization of viruses. A virus is a program which propagates its own code with possible mutations.

Definition 1 (Computer Virus). *Let B be a computable function. A virus w.r.t B is a program $\mathbf{v}$ such that $\forall \mathbf{p}, x : [\![\mathbf{v}]\!](\mathbf{p}, x) = [\![B(\mathbf{v}, \mathbf{p})]\!](x)$. Then, B is named a propagation function for the virus $\mathbf{v}$.*

Note that any virus $\mathbf{v}$ accepts infinitely many different propagation functions. As a result, the fact that $\mathbf{v}$ is a virus w.r.t some propagation function B does not imply that $\mathbf{v}$ must infect through B; it can use another propagation procedure.

Definition 1 includes the ones of Adleman and Cohen, and it handles more propagation and duplication features than the other models [4].

Definition 2 (Virus Distribution). *A virus distribution is a pair* $(\mathbf{d_v}, \mathbf{d_B})$ *of programs such that for all* i, $[\![\mathbf{d_v}]\!](i)$ *is a virus w.r.t* $[\![[\![\mathbf{d_B}]\!](i)]\!]$. $\mathbf{d_v}$ *is named* a distribution engine *and* $\mathbf{d_B}$ *is named* a propagation engine.

We refer to [24, 9] for practical details about virus distributions. As we will see later, a virus distribution can be used as a model of polymorphism. Instead of virus distributions, we speak about *virus builders* when i is intended to be a code of the behavior of the virus.

Definition 3 (Distribution builder). *A Distribution builder is a pair of programs* $\mathbf{c_v}, \mathbf{c_B}$ *such that for all* $\mathbf{h}$, $([\![\mathbf{c_v}]\!](\mathbf{h}), [\![\mathbf{c_B}]\!](\mathbf{h}))$ *is a virus distribution.*

3 Blueprint Duplication

3.1 Blueprint Virus Builder

From [4], a *blueprint virus for a function* g is a program $\mathbf{v}$ which computes g using its own code $\mathbf{v}$ and its environment $\mathbf{p}, x$. The function g can be seen as the behavior of the virus. In other words, it is a program which satisfies

$$\begin{cases} \mathbf{v} \text{ is a virus w.r.t some propagation function} \\ \forall \mathbf{p}, x : [\![\mathbf{v}]\!](\mathbf{p}, x) = g(\mathbf{v}, \mathbf{p}, x) \end{cases} \tag{4}$$

Note that a blueprint virus does not use any code of its propagation function. Clearly, the solutions of this system are provided by Kleene's recursion theorem.

Theorem 4 (Kleene's Recursion Theorem [16]). *Let* f *be a semi-computable function. There is a program* $\mathbf{e}$ *such that* $[\![\mathbf{e}]\!](x) = f(\mathbf{e}, x)$.

Corollary 5. *There is a virus builder* $(\mathbf{d_v}, \mathbf{d_B})$ *such that for any program* $\mathbf{g}$, $[\![\mathbf{d_v}]\!](\mathbf{g})$ *is a blueprint virus for* $[\![\mathbf{g}]\!]$.

Proof. We use a construction for the recursion theorem due to Smullyan [23]. It provides a fixpoint which can be directly used as a virus builder. Let $\mathbf{dg}$ and $\mathbf{d_v}$ be the respective programs.

```
dg (z,y,x){
 r := exec(z,spec(y,z,y),x);
 return r;
}
```

```
d_v (g){
 r := spec(dg,g,dg);
 return r;
}
```

```
cspec (x){
 r := spec_1;
 return r;
}
```

We observe that $[\![[\![\mathbf{d_v}]\!](\mathbf{g})]\!](\mathbf{p}, x) = g([\![\mathbf{d_v}]\!](\mathbf{g}), \mathbf{p}, x)$. Moreover, $[\![\mathbf{d_v}]\!](\mathbf{g})$ is clearly a virus w.r.t $[\![\mathbf{spec_1}]\!]$. Then, we define $\mathbf{d_B} = \mathbf{cspec}$. □

We consider a typical example of blueprint duplication which looks like the real life virus `ILoveYou`. This program arrives as an e-mail attachment. Opening the attachment triggers the attack, the infection first scans the memory for passwords and sends them back to the attacker, then the virus self-duplicates

sending itself at every address of the local address book. Here, the propagation depends on an external actor who opens the attachment.

To give a model of this scenario we need to deal with mailing processes. A mail $m = \langle @, y \rangle$ is an association of an address @ and data y. Then, we consider that the environment contains a mailbox $mb = \langle m_1, \ldots, m_n \rangle$ which is a sequence of mails. To send a mail m, we add it to the mailbox, that is to assign $mb := \texttt{cons}(m, mb)$. We suppose that an external process deals with mailing.

In the following, x denotes the local file structure, and $@bk = \langle @_1, \ldots, @_n \rangle$ denotes the local address book, a sequence of addresses. We finally introduce a $\texttt{WHILE}^+$ program **find** which searches its input for passwords and which returns them as its evaluation. The virus behavior for the scenario of `ILoveYou` is given by the following program.

```
g (v,mb,@bk,x) {
 pass := exec(find,x);
 mb := cons(cons("badguy@dom.com",pass),mb);
 y := @bk;
 while (y) {
  mb := cons(cons(hd(y),v),mb);
  y := tl(y);
 }
 return cons(mb,cons(@bk,x));
}
```

The corresponding blueprint virus is yield by the virus builder of Corollary 5.

3.2 Blueprint Distributions

A *blueprint distribution* is a virus distribution which achieves blueprint duplication. The viruses of such a distribution can use a code of the distribution engine. As a result, any member can generate the entire distribution. This idea conveys an evolving ability. More formally, a blueprint distribution $(\mathbf{d_v}, \mathbf{d_B})$ for a function g is defined by the following system.

$$\begin{cases} (\mathbf{d_v}, \mathbf{d_B}) \text{ is a virus distribution} \\ \forall i, \mathbf{p}, x : [\![[\![\mathbf{d_v}]\!](i)]\!](\mathbf{p}, x) = g(\mathbf{d_v}, i, \mathbf{p}, x) \end{cases} \tag{5}$$

The function g is thought of as the behavior of the whole distribution. The existence of blueprint distributions corresponds to a stronger form of the recursion theorem due to Case [5].

Theorem 6 (Delayed Recursion [5]). *Let f be a semi-computable function. There exists a computable function e such that $\forall x, y : [\![e(x)]\!](y) = f(\mathbf{e}, x, y)$ where $\mathbf{e}$ computes e.*

Corollary 7. *There is a distribution builder $(\mathbf{c_v}, \mathbf{c_B})$ such that for any program $\mathbf{g}$, $([\![\mathbf{c_v}]\!](\mathbf{g}), [\![\mathbf{c_B}]\!](\mathbf{g}))$ is a blueprint distribution for $[\![\mathbf{g}]\!]$.*

Proof. We use a construction close to the proof of Corollary 5. It provides a fixpoint which can be directly used as a distribution builder. We define:

```
edg (z,t,x,y) {
 e := spec(spec_3,t,z,t);
 return exec(z,e,x,y);
}
```

```
c_v (g){
 r := spec(spec_3,edg,g,edg);
 return r;
}
```

```
c_B (g){
 r := cspec;
 return r;
}
```

We observe that for any i, $[\![[\![[\![\mathbf{c_v}]\!](\mathbf{g})]\!](i)]\!](\mathbf{p},x) = g([\![\mathbf{c_v}]\!](\mathbf{g}),i,\mathbf{p},x)$. Moreover, $[\![[\![\mathbf{c_v}]\!](\mathbf{g})]\!](i)$ is a virus w.r.t $[\![[\![[\![\mathbf{c_B}]\!](\mathbf{g})]\!](i)]\!]$. □

To illustrate Theorem 7, we come back to the scenario of the virus `ILoveYou`, and we add to it mutation abilities. We introduce a `WHILE`$^+$ program **poly** which is a polymorphic engine. This program takes a program $\mathbf{p}$ and a key i, and it rewrites $\mathbf{p}$ according to i and conserving the semantics of $\mathbf{p}$. Formally, **poly** satisfies $[\![\mathbf{poly}]\!](\mathbf{p},i)$ is one-one on i and $[\![[\![\mathbf{poly}]\!](\mathbf{p},i)]\!] \approx [\![\mathbf{p}]\!]$. From a recursion theoric point of view, $[\![\mathbf{poly}]\!]$ is a padding function.

We build a virus which self-duplicates sending mutated forms of itself. With the notations of the Sect. 3.1, we consider a behavior described by the following `WHILE`$^+$ program.

```
g (d_v,i,mb,@bk,x) {
 pass := exec(find,x);
 mb := cons(cons(“badguy@dom.com”,pass),mb);
 next_key := cons(nil,i)
 virus := exec(d_v,next_key);
 mutation := exec(poly,virus,i);
 y := @bk;
 while (y) {
  mb := cons(cons(hd(y),mutation),mb);
  y := tl(y);
 }
 return cons(mb,cons(@bk,x));
}
```

We apply Corollary 7 to transform this program into a code of the corresponding distribution engine.

4 Smith Reproduction

Blueprint viruses do not use their propagation function. Now, we present a more evolved model which captures the notion of infection. Here, viruses include their propagation mechanism within their own code. As parasitic viruses do.

4.1 Smith Viruses

We define a *smith virus* as two programs $\mathbf{v}, \mathbf{B}$ which compute a behavior g according to the following system.

$$\begin{cases} \mathbf{v} \text{ is a virus w.r.t } [\![\mathbf{B}]\!] \\ \forall \mathbf{p}, x : [\![\mathbf{v}]\!](\mathbf{p}, x) = g(\mathbf{B}, \mathbf{v}, \mathbf{p}, x) \end{cases} \tag{6}$$

This class corresponds to the double recursion theorem due to Smullyan [20].

Theorem 8 (Double Recursion Theorem [20]). *Let f_1 and f_2 be two semi-computable functions. There are two programs $\mathbf{e}_1$ and $\mathbf{e}_2$ such that*

$$[\![\mathbf{e}_1]\!](x) = f_1(\mathbf{e}_1, \mathbf{e}_2, x) \quad [\![\mathbf{e}_2]\!](x) = f_2(\mathbf{e}_1, \mathbf{e}_2, x) \tag{7}$$

Corollary 9. *There is a virus builder $(\mathbf{d_v}, \mathbf{d_B})$ such that for any program $\mathbf{g}$, $[\![\mathbf{d_v}]\!](\mathbf{g}), [\![\mathbf{d_B}]\!](\mathbf{g})$ is a smith virus for $[\![\mathbf{g}]\!]$.*

Proof. We use a double fixpoint. We define the following programs.

```
dg1 (z1,z2,y1,y2,x) {
 e1 := spec(y1,z1,z2,y1,y2);
 e2 := spec(y2,z2,z2,y1,y2);
 return exec(z1,e1,e2,x);
}
```

```
dg2 (z1,z2,y1,y2,x) {
 e1 := spec(y1,z1,z2,y1,y2);
 e2 := spec(y2,z2,z2,y1,y2);
 return exec(z2,e1,e2,x);
}
```

```
pispec (g,B,v,y,p) {
 r := spec(g B,v,p);
 return r;
}
```

Let $\mathbf{d_v}$ and $\mathbf{d_B}$ be the following programs.

```
dv (g){
 r := spec(pispec,g);
 return spec(dg2,r,g,dg1,dg2);
}
```

```
dB (g){
 r := spec(pispec,g);
 return spec(dg1,r,g,dg1,dg2);
}
```

We observe that for any program $\mathbf{g}$

$$[\![[\![\mathbf{d_v}]\!](\mathbf{g})]\!](\mathbf{p}, x) = [\![[\![[\![\mathbf{d_B}]\!](\mathbf{g})]\!]([\![\mathbf{d_v}]\!](\mathbf{g}), \mathbf{p})]\!](x) = g([\![\mathbf{d_B}]\!](\mathbf{g}), [\![\mathbf{d_v}]\!](\mathbf{g}), \mathbf{p}, x) \qquad \square$$

We present how to build the parasitic virus of Sect. 2. We define

```
g (B,v,p,q,x) {
 infected_form := exec(B,v,q);
 return exec(p,infected_form,x);
}
```

The behavior $\mathbf{g}$ of the virus is the following. First, it infects a new host $\mathbf{q}$ with the virus $\mathbf{v}$ using the propagation procedure $\mathbf{B}$. Then, it executes the original host $\mathbf{p}$. This corresponds to the behavior of a parasitic virus. We obtain a smith virus using the builder of Corollary 9.

Concerning the construction of Sect. 2, the attentive reader would notice $[\![[\![\mathbf{B}]\!](\mathbf{v}, \mathbf{p})]\!] \approx [\![[\![\mathbf{spec_1}]\!](\mathbf{v}, \mathbf{id}); \mathbf{p}]\!]$ which justifies the definition of B in the example.

4.2 Smith Distributions

Smith distributions model viruses which are able to mutate their code and their propagation mechanism. A *smith distribution* $(\mathbf{d_v}, \mathbf{d_B})$ for the behavior g satisfies

$$\begin{cases} (\mathbf{d_v}, \mathbf{d_B}) \text{ is a virus distribution} \\ \forall i, \mathbf{p}, x : [\![[\![\mathbf{d_v}]\!](i)]\!](\mathbf{p}, x) = g(\mathbf{d_B}, \mathbf{d_v}, i, \mathbf{p}, x) \end{cases} \tag{8}$$

This class of distributions corresponds to the following theorem.

Theorem 10 (Delayed Double Recursion). *Let f_1 and f_2 be two semi-computable functions. There are two computable functions e_1 and e_2 such that for all x and y*

$$[\![e_1(x)]\!](y) = f_1(\mathbf{e_1}, \mathbf{e_2}, x, y) \quad [\![e_2(x)]\!](y) = f_2(\mathbf{e_1}, \mathbf{e_2}, x, y)$$
where $\mathbf{e_1}$ and $\mathbf{e_2}$ respectively compute e_1 and e_2

Corollary 11. *There is a distribution builder $(\mathbf{c_v}, \mathbf{c_B})$ such that for any program $\mathbf{g}$, $([\![\mathbf{c_v}]\!](\mathbf{g}), [\![\mathbf{c_B}]\!](\mathbf{g}))$ is a smith distribution for $[\![\mathbf{g}]\!]$.*

Proof. We define the following programs.

```
edg1 (z1,z2,t1,t2,x,y) {
 e1 := spec(spec5,t1,z1,z2,t1,t2);
 e2 := spec(spec5,t2,z1,z2,t1,t2);
 return exec(z1,e1,e2,x,y);
}
pispec′ (g,dB,dv,i,y,p) {
 r := spec(g dv,dv,i,p);
 return r;
}
```

```
edg2 (z1,z2,t1,t2,x,y) {
 e1 := spec(spec5,t1,z1,z2,t1,t2);
 e2 := spec(spec5,t2,z1,z2,t1,t2);
 return exec(z2,e1,e2,x,y);
}
```

Let $\mathbf{c_v}$ and $\mathbf{c_B}$ be the following programs.

```
cv (g){
r := spec(pispec′,g)
return spec(spec5,edg2,r,g,edg1,edg2);
}
```

```
cB (g){
r := spec(pispec′,g)
return spec(spec5,edg1,r,g,edg1,edg2);
}
```

We observe that for any program $\mathbf{g}$

$$\begin{aligned} [\![[\![[\![\mathbf{c_v}]\!](\mathbf{g})]\!](i)]\!](\mathbf{p}, x) &= [\![[\![[\![[\![\mathbf{c_B}]\!](\mathbf{g})]\!](i)]\!]([\![[\![\mathbf{c_v}]\!](\mathbf{g})]\!](i), \mathbf{p})]\!](x) \\ &= g([\![\mathbf{c_B}]\!](\mathbf{g}), [\![\mathbf{c_v}]\!](\mathbf{g}), i, \mathbf{p}, x) \end{aligned}$$ □

We enhance the virus of Sect. 4.1, adding some polymorphic abilities. Any virus of generation i infects a new host $\mathbf{q}$ with a virus of generation $\langle i, \texttt{nil} \rangle$ using the propagation procedure of generation i. Then it gives the control back to the original host $\mathbf{p}$. This behavior is illustrated by the following program.

```
g (d_B,d_v,i,p,q,x) {
 B := exec(d_B,i);
 v := exec(d_v,cons(i,nil));
 mutation := exec(poly,v,i);
 infected_form := exec(B,mutation,q);
 return exec(p,infected_form,x);
}
```

Then, we obtain the smith distribution by the builder of Corollary 11.

References

1. L. Adleman. An abstract theory of computer viruses. In *Advances in Cryptology – CRYPTO'88*, volume 403. Lecture Notes in Computer Science, 1988.
2. M. Blum. A machine-independent theory of the complexity of recursive functions. *Journal of the Association for Computing Machinery*, 14(2):322–336, 1967.
3. G. Bonfante, M. Kaczmarek, and J.-Y. Marion. Toward an abstract computer virology. In *ICTAC*, pages 579–593, 2005.
4. G. Bonfante, M. Kaczmarek, and J.-Y. Marion. On abstract computer virology from a recursion-theoretic perspective. *Journal in Computer Virology*, 1(3-4), 2006.
5. J. Case. Periodicity in generations of automata. *Theory of Computing Systems*, 8(1):15–32, 1974.
6. D. Chess and S. White. An undetectable computer virus. *Proceedings of the 2000 Virus Bulletin Conference (VB2000)*, 2000.
7. F. Cohen. *Computer Viruses*. PhD thesis, University of Southern California, January 1986.
8. F. Cohen. On the implications of computer viruses and methods of defense. *Computers and Security*, 7:167–184, 1988.
9. O. Drori, N. Pappo, and D. Yachan. New malware distribution methods threaten signature-based av. *Virus Bulletin*, pages 9–11, September 2005.
10. E. Filiol. *Computer Viruses: from Theory to Applications*. Springer-Verlag, 2005.
11. E. Filiol. Malware pattern scanning schemes secure against black-box analysis. *Journal of Computer Virology*, 2(1):35–50, 2006.
12. T. Hansen, T. Nikolajsen, J. Träff, and N. Jones. Experiments with implementations of two theoretical constructions. In *Lecture Notes in Computer Science*, volume 363, pages 119–133. Springer Verlag, 1989.
13. N. Jones. Computer implementation and applications of kleene's S-m-n and recursive theorems. In Y. N. Moschovakis, editor, *Lecture Notes in Mathematics, Logic From Computer Science*, pages 243–263. Springer Verlag, 1991.
14. N. Jones. *Computability and Complexity: From a Programming Perspective*. MIT Press, Cambridge, MA, USA, 1997.
15. N. Jones. *Constant Time Factors* Do *Matter*. MIT Press, Cambridge, MA, USA, 1997.
16. S. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.
17. M. Ludwig. *The Giant Black Book of Computer Viruses*. American Eagle Publications, 1998.
18. L. Moss. Recursion theorems and self-replication via text register machine programs. In *EATCS bulletin*, 2006.

19. P. Odiffredi. *Classical Recursion Theory*. North-Holland, 1989.
20. H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw Hill, New York, 1967.
21. M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Co. Boston, MA, USA, 1996.
22. R. Smullyan. *Recursion Theory for Metamathematics*. Oxford University Press, 1993.
23. R. Smullyan. *Diagonalization and Self-Reference*. Oxford University Press, 1994.
24. P. Ször. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.
25. K. Thompson. Reflections on trusting trust. *Communications of the Association for Computing Machinery*, 27(8):761–763, 1984.
26. J. von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, Illinois, 1966. edited and completed by A.W.Burks.
27. Z. Zuo and M. Zhou. Some further theoretical results about computer viruses. *The Computer Journal*, 47(6):627–633, 2004.
28. Z. Zuo, Q.-x. Zhu, and M.-t. Zhou. On the time complexity of computer viruses. *IEEE Transactions on information theory*, 51(8):2962–2966, August 2005.

A A concrete syntax for $\texttt{WHILE}^+$

Take $\lambda x.\underline{x} : \{\texttt{nil}, \texttt{quote}, \texttt{v}, \texttt{cons}, \texttt{hd}, \texttt{tl}, \texttt{exec}, \texttt{spec}, :=, ;, \texttt{while}, \texttt{if}\} \to \mathbb{D}$ one one. We suppose that we are given a one-one mapping from $\mathbb{V}$ to $\mathbb{D}$. The concrete syntax of $\texttt{WHILE}^+$ is defined by the following extension of $\underline{*}$. For any $d \in \mathbb{D}$, $x \in \mathbb{V}$, $e_1, e_2 \in \mathbb{E}$, $c_1, c_2 \in \mathbb{E}$,

$$
\begin{aligned}
\underline{d} &= \langle \underline{\texttt{quote}}, d \rangle \\
\underline{x} &= \langle \underline{\texttt{v}}, \underline{x} \rangle \\
\underline{\texttt{cons}(e_1, e_2)} &= \langle \underline{\texttt{cons}}, \underline{e_1}, \underline{e_2} \rangle \\
\underline{\texttt{hd}(e_1)} &= \langle \underline{\texttt{hd}}, \underline{e_1} \rangle \\
\underline{\texttt{tl}(e_1)} &= \langle \underline{\texttt{tl}}, \underline{e_1} \rangle \\
\underline{\texttt{exec}(e_0, \ldots, e_n)} &= \langle \underline{\texttt{exec}}, \underline{e_0}, \ldots, \underline{e_n} \rangle \\
\underline{\texttt{spec}(e_0, \ldots, e_n)} &= \langle \underline{\texttt{spec}}, \underline{e_0}, \ldots, \underline{e_n} \rangle \\
\underline{x := e_1} &= \langle \underline{:=}, \underline{\texttt{v}}, \underline{e}, \underline{e_1} \rangle \\
\underline{c_1; c_2} &= \langle \underline{;}, \underline{c_1}, \underline{c_2} \rangle \\
\underline{\texttt{while}(e_1)\{c_1\}} &= \langle \underline{\texttt{while}}, \underline{e}, \underline{c_1} \rangle \\
\underline{\texttt{if}(e)\{c_1\}\texttt{else}\{c_2\}} &= \langle \underline{\texttt{if}}, \underline{e_1}, \underline{c_1}, \underline{c_2} \rangle
\end{aligned}
$$

Let $\texttt{v}_0$ be an element of $\mathbb{V}$. By convention for any element $\mathbf{p} \in \mathbb{D}$ which is not a concrete syntax of any $\texttt{WHILE}^+$ program, we define $\mathbf{p}$ as the concrete syntax of $(\texttt{v}_0)\{\texttt{v}_0 := \texttt{nil}; \texttt{return nil};\}$.

B Semantics of $\texttt{WHILE}^+$

We note π_1 and π_2 the projections associated to $\langle\ ,\ \rangle$. By convention $\pi_1(\texttt{nil}) = \pi_2(\texttt{nil}) = \texttt{nil}$. Finite sequences are built by repeated applications of the pairing function: $\langle x_1, \ldots, x_n \rangle = \langle x_1, \langle x_2, \langle \ldots, x_n \rangle \ldots \rangle \rangle$.

A store is a mapping $\sigma : \mathbb{V} \to \mathbb{D}$. The set of stores is noted $\mathbb{S}$. For any $x \in \mathbb{V}$ the notation $\sigma[x \mapsto d]$ denotes the function σ' such that $\sigma'(x) = d$ and for any $y \in \mathbb{V}$ such that $y \neq x$, $\sigma'(y) = \sigma(y)$. By extension, for any $x_1, \ldots, x_k$, $[x_1 \mapsto d_1, \ldots, x_k \mapsto d_k]$ denotes the store σ such that $\forall i \leq k : \sigma(x_i) = d_i$ and $\forall i > k : \sigma(x_i) = \texttt{nil}$.

The semantics is defined by three functions, $\mathcal{E} : \mathbb{E} \to (\mathbb{S} \to \mathbb{D})$ for expressions, $\mathcal{C} : \mathbb{C} \to (\mathbb{S} \to \mathbb{S})$ for commands and $\mathcal{P} : \mathbb{P} \to (\mathbb{D} \to \mathbb{D})$ for $\texttt{WHILE}^+$ programs.

Evaluation of expressions. For any $d \in \mathbb{D}$, $x \in \mathbb{V}$, $e_1, e_2 \in \mathbb{E}$, $\sigma \in \mathbb{S}$

$$\begin{aligned}
\mathcal{E}_d(\sigma) &= d \\
\mathcal{E}_x(\sigma) &= \sigma(x) \\
\mathcal{E}_{\texttt{cons}(e_1,e_2)}(\sigma) &= \langle \mathcal{E}_{e_1}(\sigma), \mathcal{E}_{e_2}(\sigma) \rangle \\
\mathcal{E}_{\texttt{hd}(e_1)}(\sigma) &= \pi_1(\mathcal{E}_{e_1}(\sigma)) \\
\mathcal{E}_{\texttt{tl}(e_1)}(\sigma) &= \pi_2(\mathcal{E}_{e_1}(\sigma)) \\
\mathcal{E}_{\texttt{exec}(e,e_1,\ldots,e_n)}(\sigma) &= [\![\mathcal{E}_e(\sigma)]\!](\langle \mathcal{E}_{e_1}(\sigma), \ldots, \mathcal{E}_{e_n}(\sigma) \rangle)
\end{aligned}$$

Evaluation of commands. For any $e \in \mathbb{E}$, $c_1, c_2 \in \mathbb{C}$, $\sigma \in \mathbb{S}$

$$\begin{aligned}
\mathcal{C}_{x\,:=\,e}(\sigma) &= \sigma[x \mapsto \mathcal{E}_e(\sigma)] \\
\mathcal{C}_{c_1;\,c_2}(\sigma) &= \mathcal{C}_{c_2}(\mathcal{C}_{c_1}(\sigma)) \\
\mathcal{C}_{\texttt{while}(e)\{c_1\}}(\sigma) &= \begin{cases} \sigma & \text{if } \mathcal{E}_e(\sigma) = \texttt{nil} \\ \mathcal{C}_{\texttt{while}(e)\{c_1\}}(\mathcal{C}_c(\sigma)) & \text{otherwise} \end{cases} \\
\mathcal{C}_{\texttt{if}(e)\{c_1\}\texttt{else}\{c_2\}} &= \begin{cases} \mathcal{C}_{c_2}(\sigma) & \text{if } \mathcal{E}_e(\sigma) = \texttt{nil} \\ \mathcal{C}_{c_1}(\sigma) & \text{otherwise} \end{cases}
\end{aligned}$$

Evaluation of programs. For any $d \in \mathbb{D}$, $x_1, \ldots, x_k \in \mathbb{V}$,$e \in \mathbb{E}$, $c \in \mathbb{C}$

$$\begin{aligned}
&\mathcal{P}_{(x_1,\ldots,x_k)\{c;\,\texttt{return}\ e;\,\}}(d) = \mathcal{E}_e(\mathcal{C}_c([x_1 \mapsto d_1, \ldots, x_k \mapsto d_k])) \\
&\text{where } d_1 = \pi_1(d),\ \forall 1 < i < k : d_k = \pi_1(\pi_1^{k-1}(d)) \text{ and } d_K = \pi_1^k(d)
\end{aligned}$$

We define $[\![\mathbf{p}]\!] \approx \mathcal{P}_{\texttt{w}}$ where $\mathbf{p}$ is the concrete syntax of $\texttt{w}$. The programming language $[\![\ ]\!]$ can be shown acceptable, see [14]. Then, there is a computable function S_n such that $[\![S_n(\mathbf{p}, x_1, \ldots, x_n)]\!](y) = [\![\mathbf{p}]\!](x_1, \ldots, x_n, y)$. We define $\mathcal{E}_{\texttt{spec}(e,e_1,\ldots,e_n)}(\sigma) = S_n(\mathcal{E}_e(\sigma), \mathcal{E}_{e_1}(\sigma), \ldots, \mathcal{E}_{e_n}(\sigma))$ and we extend $\mathcal{E}$, $\mathcal{C}$, $\mathcal{P}$, $[\![\ ]\!]$, accordingly.