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
Nancy-Université - 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 constructions are essentially based on Kleene's second recursion theorem. We show that we can classify viruses as solutions of fixed point equations which are obtained from different versions of Kleene's second recursion theorem. This lead us to consider four classes of viruses which various polymorphic features. We propose to use virus distribution in order to deal with mutations.
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 [9], 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]. Since viruses are essentially self-replicating programs, we see that virus programming methods are an attempt to answer to von Neumann's question [22].

> 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?

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 [23] and they analyzed the time complexity of viruses [24].

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 characterized two kinds of viruses, blueprint and smith viruses, and we proved constructively their existence. This work proposes to go further, introducing a notion of distribution to take into account polymorphism or metamorphism. We define four kinds of viruses:

1. A blueprint virus is a virus, which reproduces by just duplicating its code.
2. An evolving blueprint virus is a virus, which can mutate when it duplicates by modifying its code. Evolving blueprint viruses are generated by a disbution engine.
3. A smith virus is a blueprint virus which can use its propagation function directly to reproduce.
4. Lastly, we present Smith distribution. A virus generating by a Smith distribution can mutate its code like evolving blueprint viruses, but also mutate its propagation function.

We show that each category is closely linked to a corresponding form of the recursion theorem, given a rational taxonomy of viruses. So recursion theorems play a key role in *constructions* of viruses, which is worth to mention. Indeed, and despite the works [11, 12], recursion theorems are used essentially to prove "negative" results such as the constructions of undecidable or inseparable sets, see [19] for a general reference, or such as Blum's speed-up theorem [2].

Lastly, we switch to a simple programming language named $\texttt{WHILE}^+$ to illustrate the fact that our constructions lives in the programming world. Actually, we follow the ideas of the experimentation of the iteration theorem and of the recursion theorem, which are developed in [11, 12] by Jones et al. and very recently by Moss in [16].

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 $\texttt{nil}$ and a pairing mechanism $\langle\ ,\ \rangle$. The syntax of $\texttt{WHILE}^+$ is given by the following grammar from a set of variables $\mathbb{V}$:

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

A $\texttt{WHILE}^+$ program $\mathbf{p}$ is defined as follows $\mathbf{p}(\mathbb{V}_1, \ldots, \mathbb{V}_n)\{\mathbb{C}; \texttt{return}\ \mathbb{E};\}$. A program $\mathbf{p}$ computes a function $[\![\mathbf{p}]\!]$ from $\mathbb{D}^n$ to $\mathbb{D}$.

We suppose that we are given a concrete syntax of $\texttt{WHILE}^+$, that is an encoding of programs by binary trees of $\mathbb{D}$. From now on, when the context is clear,

we do not make any distinction between a program and its concrete syntax. And we make no distinction between programs and data.

For convenience, we have a built-in self-interpreter $\texttt{exec}_n$ of $\texttt{WHILE}^+$ programs which satisfies :

$$[\![\texttt{exec}_n]\!](\mathbf{p}, x_1, \ldots x_n) = [\![\mathbf{p}]\!](x_1, \ldots x_n)$$

In the above equation, the notation $\mathbf{p}$ means the concrete syntax of the program $\mathbf{p}$.

We also use a built-in specializer $\texttt{spec}_n$ which satisfies:

$$[\![[\![\texttt{spec}_m]\!](\mathbf{p}, x_1, \ldots x_m)]\!](x_{m+1}, \ldots, x_n) = [\![\mathbf{p}]\!](x_1, \ldots x_n)$$

We may omit the subscpript n which indicates the number of arguments of an interpreter or a specializer.

The use of an interpreter and of a specializer is justified by Jones who showed in [13] 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 A Computer Virus representation

We propose the following scenario in order to represent viruses. When a program $\mathbf{p}$ is executed within an environment x, the evaluation of $[\![\mathbf{p}]\!](x)$, if it halts, is a new environment. This process may be then repeated by replacing x by the new computed environment. The entry x is thought of as a finite sequence $\langle x_1, \ldots, x_n \rangle$ which represents files and accessible parameters.

Typically, a program **copy** which duplicates 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 $\langle \mathbf{p}, \mathbf{p}, x \rangle$ in which $\mathbf{p}$ is copied.

Next consider an example of *parasitic virus*. 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. For more details we refer to the virus writing manual of Ludwig [15]. A parasistic virus is a program $\mathbf{v}$ which works on an environment $\langle \mathbf{p}, \mathbf{q}, x \rangle$. The infected form of $\mathbf{p}$ is $B(\mathbf{v}, \mathbf{p})$ where B is a propagation function which specifies how a virus contaminates a file. Here, the propagation function B can be for example a program code concatenation function. So, we have a first "generic" equation: $[\![\mathbf{v}]\!](\mathbf{p}, \langle \mathbf{q}, x \rangle) = [\![B(\mathbf{v}, \mathbf{p})]\!](\langle \mathbf{q}, x \rangle)$. Following the description of a parasitic virus, $\mathbf{v}$ computes the infected form $B(\mathbf{v}, \mathbf{q})$ and then executes $\mathbf{p}$. This means that the following equation also holds: $[\![\mathbf{v}]\!](\mathbf{q}, x) = [\![\mathbf{p}]\!](B(\mathbf{v}, \mathbf{q}), x)$. A parasistic virus is defined by the two above equations.

More generally, the construction of viruses lies in the resolution of fixed point equations such as the ones above in which $\mathbf{v}$ and B are unknowns. The existence of solutions of such systems is provided by Kleene's recursion theorem. From this observation and following [4], we propose the following virus representation:

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* $\mathbf{v}$.

This definition includes the ones of Adleman and Cohen, and it handles more propagation and duplication features than the other models [4]. However, it is worth to notice that the existence of a virus $\mathbf{v}$ w.r.t a given propagation function B is constructive. This is a key difference since it allows to build viruses by applying fixed point constructions given by proofs of recursion theorems.

A motivation behind the choice of $\texttt{WHILE}^+$ programming language is the fact that there is no self-referential operator, like \$0 in bash, which returns a copy of the program concrete syntax. Indeed, we present below virus construction without this feature. This shows that even if there is no self-referential operator, there are still viruses. Now, viruses should be more efficient if such operators are present. Of course, a seminal paper on this subject is [21].

3 Blueprint Duplication

3.1 Blueprint distribution engine

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 virus specification function. A blueprint virus for a function g is a program $\mathbf{v}$ which satisfies

$$\begin{cases} \mathbf{v} \text{ is a virus w.r.t some propagation function} \\ \forall \mathbf{p}, x : [\![\mathbf{v}]\!](\mathbf{p}, x) = g(\mathbf{v}, \mathbf{p}, x) \end{cases} \tag{1}$$

Note that a blueprint virus does not use any code of its propagation function, unlike smith viruses that we shall see shortly. The solutions of this system are provided by Kleene's recursion theorem.

Theorem 2 (Kleene's Recursion Theorem [14]). *Let f be a semi-computable function. There is a program* $\mathbf{e}$ *such that* $[\![\mathbf{e}]\!](x) = f(\mathbf{e}, x)$.

Definition 3 (Distribution engine). *A distribution engine is a program* $\mathbf{d_v}$ *such that for every virus specification program* $\mathbf{g}$, $[\![\mathbf{d_v}]\!](\mathbf{g})$ *is a virus w.r.t a fixed and given a propagation function B.*

Theorem 4. *There is a distribution engine* $\mathbf{d_v}$ *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 [20]. It provides a fixpoint which can be directly used as a distribution engine. We define $\mathbf{d_v}$ thanks to the concrete syntax of $\mathbf{dg}$ as follows:

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

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

We observe that $[\![[\![\mathbf{d_v}]\!](\mathbf{g})]\!](\mathbf{p}, x) = [\![\mathbf{g}]\!]([\![\mathbf{d_v}]\!](\mathbf{g}), \mathbf{p}, x)$. Moreover, $[\![\mathbf{d_v}]\!](\mathbf{g})$ is clearly a virus w.r.t to the propagation function $[\![\texttt{spec}]\!]$. □

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.

To represent 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 $mb := \texttt{cons}(m, mb)$. 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 mb;
}
```

From the virus specification program $\mathbf{g}$, we generate the blueprint virus $[\![\mathbf{d_v}]\!](\mathbf{g})$.

3.2 Distributions of evolving blueprint viruses

An *evolving blueprint virus* is a virus, which can mutate but the propagation function remains the same. Here, we describe a distribution engine for which the specification of a virus can use the code of its own distribution engine. Thus, we can generate evolved copies of a virus. Formally, given a virus specification function g, a distribution of evolving blueprint viruses is a program $\mathbf{d_v}$ satisfying:

$$\begin{cases} \mathbf{d_v} \text{ is a distribution engine} \\ \forall i, \mathbf{p}, x : [\![[\![\mathbf{d_v}]\!](i)]\!](\mathbf{p}, x) = g(\mathbf{d_v}, i, \mathbf{p}, x) \end{cases} \tag{2}$$

The existence of blueprint distributions corresponds to a stronger form of the recursion theorem, which was first proved by Case [5].

Theorem 5 (Explicit recursion [4]). *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.*

Definition 6 (Distribution engine builder). *A builder of distribution engine is a program $\mathbf{c_v}$ such that for every virus specification program $\mathbf{g}$, $[\![\mathbf{c_v}]\!](\mathbf{g})$ is a distribution engine.*

Theorem 7. *There is a builder of distribution engine $\mathbf{c_v}$ such that for any program $\mathbf{g}$, $[\![\mathbf{c_v}]\!](\mathbf{g})$ is a distribution of evolving blueprint viruses for some fixed propagation function B.*

Proof. We define

```
edg (z,t,i,y,x) {
 e := spec(spec3,t,z,t);
 return exec(z,e,i,y,x);
}
```

```
cv (g){
 r := spec(spec3,edg,g,edg);
 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 $[\![\mathtt{spec}]\!]$. □

To illustrate Theorem 7, we come back to the scenario of the virus `ILoveYou`, and we add to it mutation abilities. We introduce a $\mathtt{WHILE}^+$ 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, conserving the semantics of $\mathbf{p}$. That is, **poly** satisfies $[\![\mathbf{poly}]\!](\mathbf{p}, i)$ is one-one on i and $[\![[\![\mathbf{poly}]\!](\mathbf{p}, i)]\!] \approx [\![\mathbf{p}]\!]$.

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 $\mathtt{WHILE}^+$ program.

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

We apply Theorem 7 to transform this program into a code of the corresponding distribution engine. So, $[\![[\![\mathbf{c_v}]\!](\mathbf{g})]\!](i)$ is a copy indexed by i of the evolving blueprint virus specified by $\mathbf{g}$.

4 Smith Reproduction

4.1 Smith Viruses

We define a *smith virus* as two programs $\mathbf{v}, \mathbf{B}$ which is defined w.r.t a virus specification function g according to the following system.

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

The class of smith viruses is obtained by the double recursion theorem due to Smullyan [18] as a solution to the above equations.

Theorem 8 (Double Recursion Theorem [18]). *Let f_1 and f_2 be two semi-computable functions. There are two programs $\mathbf{e}_1$ and $\mathbf{e}_2$ such that*

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

We extend the previous definition of engine distribution to propagation engine as follows.

Definition 9 (Virus Distribution). *A virus distribution is a pair $(\mathbf{d_v}, \mathbf{d_B})$ of programs such that for every virus specification $\mathbf{g}$, $[\![\mathbf{d_v}]\!](\mathbf{g})$ is a virus w.r.t $[\![[\![\mathbf{d_B}]\!](\mathbf{g})]\!]$. As previously, $\mathbf{d_v}$ is named* a distribution engine *and $\mathbf{d_B}$ is named* a propagation engine.

Theorem 10. *There is a virus distribution $(\mathbf{d_v}, \mathbf{d_B})$ 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 define the following programs with a double fixed point.

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

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

and

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

Then, 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. The virus specification function $\mathbf{g}$ of the virus is the following.

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

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 Theorem 10.

4.2 Smith Distributions

Smith distributions generate viruses which are able to mutate their code and their propagation mechanism. A *smith distribution* $(\mathbf{d_v},\mathbf{d_B})$ w.r.t the virus specification program g satisfies

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

The class of Smith distributions is defined as the solutions of this double recursion theorem.

Theorem 11 (Double explicit 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.

Definition 12 (Distribution builder). *A Distribution builder is a pair of programs $\mathbf{c_v},\mathbf{c_B}$ such that for every virus specification program $\mathbf{g}$, $([\![\mathbf{c_v}]\!](\mathbf{g}),[\![\mathbf{c_B}]\!](\mathbf{g}))$ is a virus distribution.*

Theorem 13. *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,i,y,x) {
 e1 := spec(spec5,t1,z1,z2,t1,t2);
 e2 := spec(spec5,t2,z1,z2,t1,t2);
 return exec(z1,e1,e2,i,y,x);
}
and
pispec′ (g,db,dv,i,y,p) {
 r := spec(g,db,dv,i,p);
 return r;
}
```

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

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 $\mathbf{g}$

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

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 next generation 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 (db,dv,i,p,⟨q, x⟩) {
 B := exec(db,i);
 v := exec(dv,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 Theorem 13.

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. E. Filiol. *Computer Viruses: from Theory to Applications*. Springer-Verlag, 2005.
10. E. Filiol. Malware pattern scanning schemes secure against black-box analysis. *Journal of Computer Virology*, 2(1):35–50, 2006.
11. 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.
12. 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.
13. N. Jones. *Constant Time Factors* Do *Matter*. MIT Press, Cambridge, MA, USA, 1997.
14. S. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.
15. M. Ludwig. *The Giant Black Book of Computer Viruses*. American Eagle Publications, 1998.
16. L. Moss. Recursion theorems and self-replication via text register machine programs. In *EATCS bulletin*, 2006.
17. P. Odiffredi. *Classical Recursion Theory*. North-Holland, 1989.
18. H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw Hill, New York, 1967.
19. R. Smullyan. *Recursion Theory for Metamathematics*. Oxford University Press, 1993.
20. R. Smullyan. *Diagonalization and Self-Reference*. Oxford University Press, 1994.
21. K. Thompson. Reflections on trusting trust. *Communications of the Association for Computing Machinery*, 27(8):761–763, 1984.
22. J. von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, Illinois, 1966. edited and completed by A.W.Burks.
23. Z. Zuo and M. Zhou. Some further theoretical results about computer viruses. *The Computer Journal*, 47(6):627–633, 2004.
24. 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.