Des fondements de la virologie informatique vers une immunologie formelle

Matthieu Kaczmarek

Nancy-Université INPL

Département de formation doctorale en informatique

Institut National Polytechnique de Lorraine

École doctorale IAEM Lorraine

Des fondements de la virologie informatique vers une immunologie formelle

THÈSE

présentée et soutenue publiquement le 3 décembre 2008

pour l'obtention du

Doctorat de l'Institut National Polytechnique de Lorraine

(spécialité informatique)

par

Matthieu Kaczmarek

Composition du jury

Rapporteurs :	José M. Fernandez	Professeur associé, Polytechnique Montréal
	Serge Grigorieff	Professeur, Université Paris 7
Examinateurs :	Guillaume Bonfante	Maître de conférences, Nancy-Université – INPL
	Olivier Festor	Directeur de recherche, LORIA
	Eric Filiol	Professeur, ESIEA
	Jean-Yves Marion	Professeur, Nancy-Université – INPL
	Jean-Marc Steyaert	Professeur, Ecole Polytechnique

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Loria

Remerciements

Je voudrais remercier le centre national de la recherche scientifique et la région lorraine pour le soutient financier qui m'a permis de mener cette thèse. Je remercie ausi l'agence national pour la recherche qui par l'intermédiaire de l'action de recherche en amont VIRUS a participé au soutient financié de projets liés à cette thèse.

J'aimerais remercier mes directeur et co-directeur de thèse, Jean-Yves Marion et Guillaume Bonfante, pour leur patience et leur écoute lors de nos entretiens. J'espère que les liens scientifiques et amicaux construits durant ces trois années perdurerons.

Je voudrais aussi remercier Eric Filiol, non seulement pour le temps qu'il m'a accordé mais aussi pour l'énergie qu'il a développé pour relancer et soutenir la virologie informatique.

Merci à Romain et Emmanuel pour leur aciduité aux pauses café, à Octave pour sa solicitude face à nos voisins de chambré et plus généralement aux membres de l'équipe CARTE.

Plus personnellement, je remercie mes parents et mon frère qui m'ont permis de devenir celui que je suis. Enfin, je remercie Elise pour ce bonheur au quotidien et Léo-Paul pour être le plus adorable des bébés.

Table des matières

Introduction

Les virus informatiques constituent un enjeu de société dont les aspects économiques se mesurent en milliard de dollars. Selon les *Consumer Reports* de septembre 2008, les attaques par programmes malicieux et par ameçonnage ont coûté 8,5 milliards de dollars aux Etats-Unis sur une période de deux années. Au-delà de l'aspect économique, ces attaques mettent également en jeu la pérennité des systèmes d'information sur lesquels repose notre société, comme la confidentialité des services de l'administration ou encore le vote électronique.

Si les risques et les conséquences des infections informatiques sont bien évalués, il n'en va pas de même de leur identité, de leur mécanisme et de leur nature. Cette thèse a pour objet la virologie informatique et vise à pallier ces manques. Cette discipline, ouverte par Cohen et Adleman à la fin des années 80, est encore peu étudiée. Il s'agit d'examiner les virus informatiques d'un point de vue abstrait en laissant de coté les détails techniques non significatifs. L'objectif est de mettre en évidence les éléments essentiels à leur construction.

Les virus se définissent et se comprennent aisément à partir des racines de l'informatique fondamentale, à savoir le théorème d'itération et le théorème de récursion de Kleene. Ce premier théorème montre comment spécialiser un programme, quant au second il permet de construire des programmes auto-référents, c'est très certainement l'un des résultats les plus profonds de l'informatique fondamentale. En particulier, le théorème de récursion a souvent été associé au principe d'auto-reproduction de von Neumann [Rog67]. A partir de ces piliers, Adleman [Adl88] modélise les virus informatiques d'une manière élégante et pragmatique. Comme le montre l'ouvrage de Filiol [Fil05], la plupart des techniques antivirales actuelles sont inspirées des travaux de Cohen et Adleman. On comprend qu'une telle étude théorique est nécessaire à la conception de solutions antivirales.

Au delà de la dimension théorique, la virologie informatique nécessite une recherche empirique. D'une part, le développement d'outils de détection et de protection passe par des observations pratiques, d'autre part la validation des réponses théoriques apportées devra être réalisée expérimentalement. Ces deux phases de recherche nécessitent la manipulation de programmes potentiellement dangereux pour les systèmes informatiques. Ainsi ce travail de thèse devra se confronter à la problématique de l'expérimentation en milieu protégé.

Résumé du manuscrit

Cette thèse se décompose en trois grands thèmes : la formalisation de la virologie informatique, l'élaboration de protections contre l'auto-reproduction et le problème de la détection des programmes malicieux. Chacun de ces thèmes fera l'objet d'une partie du manuscrit. Une partie préliminaire introduit les concepts d'auto-références nécessaires à la lecture de ce manuscrit.

Auto-référence et calculabilité.

Chapitre 1. Ce chapitre introduit les fondements de la virologie informatique d'une manière accessible et ludique. En premier lieux, il montre comment construire des objets auto-référents sans l'aide de pointeur. Dans un second temps, il met en évidence les relations qu'entretiennent les mécanismes d'auto-référence et la virologie informatique.

Chapitre 2. L'étude des virus informatiques repose sur la notion de programme. Pour la définir, nous recourons à un langage de programmation particulier nommé `While`. Nous empruntons cette vision moderne de la calculabilité à Jones [Jon97]. Au delà de sa dimension constructive, cette approche a l'avantage d'être pédagogique. En effet, le langage `While` est simple à présenter et à comprendre. De plus, ce formalisme permet de représenter les programmes à l'intérieur du domaine de calcul. Nous pouvons ainsi construire des transformateurs de programmes, c'est à dire des programmes modifiant ou construisant d'autres programmes. Cet aspect est essentiel à la virologie informatique puisqu'un virus informatique peut être vu comme un programme transformant un programme hôte en un programme infecté.

Chapitre 3. Nous complétons le chapitre précédent en montrant que notre formalisme est générique. Pour cela, nous nous appuyons sur le théorème d'isomorphisme de Rogers, il nous assure que, à un isomorphisme près, nos constructions sont reproductibles dans la majorité des modèles de calcul raisonnables. L'exposition de ce résultat nous amène à présenter les notions d'auto-interprétation et de spécialisation, deux objets fondamentaux en virologie informatique. Dans une seconde partie, nous exposons le second théorème de récursion de Kleene qui fournit une construction systématique de programmes auto-référents. Nous montrons que les programmes produits sont relativement efficaces et nous présentons quelques applications. Nous nous attardons sur ce théorème car il est au centre de la notion de virus informatique.

Virologie informatique. Nous proposons une formalisation de la virologie informatique. Elle est construite sur les travaux fondateurs de la discipline. Nous obtenons un formalisme souple où le théorème de récursion prend un rôle central.

Chapitre 4. A travers la présentation des travaux de von Neumann [vN66], Cohen [Coh86, Coh88] et Adleman [Adl88], nous commentons le chemin théorique

suivit par la virologie informatique. Ce parcours met en avant les liens qu'entretiennent cette discipline et les modèles de calcul et il permet d'identifier les caractéristiques essentielles des virus informatiques. Les travaux de von Neumann nous montrent qu'un programme auto-reproducteur peut être construit par un mécanisme de diagonalisation. En évoquant les boucles de Codd et Langton, nous soulignons que la présence d'un auto-interpréteur n'est pas nécessaire à l'auto-reproduction. L'étude du formalisme de Cohen nous permet d'identifier trois caractéristiques essentielles à la modélisation des virus informatiques. (a) Un virus est un programme. (b) Un virus peut infecter d'autres programmes. (c) La forme d'un virus peut évoluer au cours de sa reproduction. L'approche d'Adleman montre que la notion de virus informatique est indépendante du modèle de calcul considéré. De plus, il propose une première formalisation de la notion d'infection.

Chapitre 5. En s'appuyant sur les spécificités soulignées au chapitre précédent, nous proposons une unification des différents formalismes de la virologie informatique. Un virus est défini comme la solution d'un système d'équations à point fixe. Cette approche met en avant une dualité entre le code viral et la propriété fonctionnelle décrivant sa propagation. Le théorème de récursion prend un rôle central dans ce formalisme puisqu'il permet de construire des solutions du système d'équations, ainsi ce théorème prend la forme d'un compilateur de virus. Nous approfondissons cette étude en décrivant deux mécanismes de reproduction, le premier nommé *blueprint* repose sur la duplication d'un code viral et le second nommé *Smith* utilise une notion plus abstraite de propagation.

Chapitre 6. Dans un premier temps nous replaçons la notion de mutation dans le cadre de la calculabilité en la liant au lemme de padding. Ensuite, nous modélisons la notion de virus polymorphe comme un ensemble de solutions des systèmes d'équations présentés au chapitre précédent, chacune de ces solutions caractérisant une mutation du virus. Ainsi nous exposons des versions polymorphes des mécanismes de reproduction blueprint et Smith. En présentant la construction de ces virus sous la forme d'une résolution de systèmes d'équations, nous renforçons les liens entre théorèmes de récursion et constructions virales.

Méthodes de protection. En s'appuyant sur le formalisme présenté dans la partie précédente, nous étudions des stratégies de protection visant à interdire les constructions virales.

Chapitre 7. Nous menons dans le contexte des langages de programmation un travail comparable à celui réalisé par von Neumann [vN66] dans le domaine des automates cellulaires. Nous nous intéressons aux briques élémentaires suffisant à la construction de virus informatiques. Les chapitres précédents nous permettent d'affirmer qu'il existe des virus informatiques de manière intrinsèque dans tout modèle de calcul raisonnable. Ce chapitre approfondit ce point en

étudiant des langages ne satisfaisant pas la notion d'acceptabilité. En cela, il répond à l'une des interrogations de Cohen [Coh89] qui propose de limiter les capacités de calcul au point qu'il ne soit plus possible qu'un programme puisse s'auto-reproduire. Nous montrons qu'un langage de programmation aux capacités de calcul très limitées, possède l'ensemble des constructions virales. Nous expliquons ainsi que l'auto-reproduction et les capacités de calcul n'entretiennent pas une relation triviale. Cette étude nous permet de comprendre que la conception de systèmes immunisés doit aussi prendre en compte la représentation concrète des programmes.

Chapitre 8. Nous proposons deux stratégies de protections fondées sur une relation entre sémantique et syntaxe concrète. A cette fin nous utilisons un outil de l'informatique théorique reliant ces deux aspects : la complexité de Kolmogorov. La première stratégie sollicite un mécanisme de compression qui vise à réduire l'espace disponible aux virus pour s'introduire à l'intérieur d'un hôte. Cela nous permet de borner la complexité des infections parasites. La seconde stratégie est fondée sur le contrôle des flux d'information, elle vise à circonscrire les infections virales. Le principe repose sur une hiérarchisation des données à laquelle est adjoint un langage de programmation satisfaisant une propriété d'intégrité. De manière intuitive, cette propriété interdit les flux d'information depuis les niveaux inférieurs vers les niveaux supérieurs. De ce fait, si un certain niveau du système est infecté, cette infection est circonscrite aux niveaux inférieurs. Les autres niveaux restant inaccessibles et immunisés. Enfin, nous montrons comment instaurer une telle politique par un typage du langage `While`.

Détection. Dans cette partie, nous étudions la problématique de la détection des virus informatiques. Dans un premier temps, nous prenons un point de vue abstrait en nous appuyant sur notre formalisation de la virologie informatique. Puis nous considérons des aspects plus pratiques en décrivant l'architecture d'un détecteur de programmes malicieux conçu durant cette thèse.

Chapitre 9. En nous appuyant sur notre formalisation de la virologie informatique, nous identifions des scénarios précis d'infections virales. Nous en déduisons des stratégies de détection puis, dans le cadre de la hiérarchie arithmétique, nous évaluons les capacités de calcul associées à la mise en place de ces stratégies. Suivant le type d'infection informatique, le problème de la détection consiste à identifier soit les codes viraux, soit les programmes infectés. On retrouve ainsi les mécanismes blueprint et Smith étudiés dans les chapitres précédents.

Chapitre 10. Nous concluons ce manuscrit en annonçant une voie de recherche qui pourrait être entreprise par la suite. Nous présentons une stratégie de détection des programmes malicieux à mi chemin entre la détection syntaxique et la détection sémantique. Nous nommons cette approche *détection morphologique*,

l'idée étant de reconnaître la forme des programmes malicieux. A l'inverse de la détection syntaxique, nous ne considérons pas un programme comme une simple chaîne de caractères. Nous prenons en compte les liens sémantiques entre les différentes parties du programme afin d'obtenir un objet de plus grande dimension. Pour le moment, notre étude se limite au flot de contrôle mais nous pensons ajouter d'autres critères sémantiques par la suite.
L'un de nos objectifs est d'établir une méthode efficace : nous présentons un détecteur travaillant en temps cubique et nous soupçonnons l'existence d'un algorithme de complexité inférieure. De plus nous utilisons un automate d'arbres, ainsi nous bénéficions de la propriété de Myhill-Nerode qui garantit une efficacité optimale relativement à la représentation des signatures.
Afin de valider cette approche nous avons entrepris des expériences sur des échantillons de tailles moyennes. Les résultats sont prometteurs : avec une procédure automatique de construction des signatures nous obtenons un taux de faux positifs inférieur à 0.1%.

Projets scientifiques

En plus des travaux universitaires, cette thèse s'est inscrite dans deux actions importantes : l'atelier TCV et le laboratoire de haute sécurité.

L'atelier TCV. La création de l'atelier *International Workshop on the Theory of Computer Viruses* a été motivée par le nombre limité d'articles théoriques ayant traité de la virologie informatique ces vingt dernières années. Comme cela a été observé dans de nombreux domaines, une meilleure compréhension théorique est susceptible d'apporter de nouvelles réponses. L'objectif de cette action est d'utiliser les outils de l'informatique fondamentale afin de définir une virologie informatique abstraite.

L'auteur a participé à l'organisation des éditions 2005, 2006 et 2007 qui se sont déroulés dans les locaux du LORIA à Nancy. Ces ateliers ont pour but de soutenir une recherche théorique sur les menaces liées aux virus, aux vers, aux programmes espions et aux programmes malicieux. Ils ont fourni un lieu privilégié d'échanges sur ces thèmes. L'édition 2009 de cette atelier se déroulera à Berlin et sera associée à la conférence EICAR.

Le laboratoire de haute sécurité. Les expériences menées dans le cadre de la conception du détecteur morphologique a mis en évidence un manque d'infrastructure dans le domaine de la virologie informatique. Le laboratoire de haute sécurité permettra de pallier ces manques. Les applications envisagées sont le déploiement de systèmes d'attaque et de défense contre des programmes malveillants, l'utilisation de techniques virales pour développer de nouvelles technologies, la détection

de failles, l'audit de sécurité et la certification de systèmes. Ce laboratoire est essentiel à la continuité des travaux de recherche sur la virologie informatique. Il vise à mettre en place des procédés s'articulant autour de deux objectifs.

Le premier objectif est la définition d'un plan de supervision des réseaux ayant comme paradigmes les vers informatiques et les programmes malicieux. L'idée est d'aborder la gestion des infections informatiques par une approche épidémiologique. Ainsi, il est possible d'envisager des mécanismes de surveillance et de supervision fondés sur les protocoles de la santé publique. Cela implique le déploiement d'une infrastructure qui centralise les moyens de surveillance et de supervision des infections informatiques et de leur développement.

Le deuxième objectif est la mise en œuvre de nouveaux mécanismes de défense contre les programmes malicieux. Ces mécanismes visent à extraire une représentation abstraite d'un programme qui s'appuie sur le flot de contrôle. Le laboratoire de haute sécurité permettra de valider expérimentalement cette approche sans craindre la fuite de programme malicieux. Un tel projet nécessite une grande expertise afin de garantir le confinement des programmes malveillants utilisés. En effet, la moindre fuite pourrait mener à une propagation incontrôlée.

Auto-référence et calculabilité

1 Introduction à la virologie informatique

Ce premier chapitre vise à mettre en évidence les relations qu'entretiennent les mécanismes d'auto-référence et la virologie informatique. Nous tâcherons de réaliser cet objectif en conservant un formalisme accessible au néophyte. Une large partie des constructions suivantes est empruntée au premier chapitre de l'excellent ouvrage de Smullyan [Smu94]. Nous replaçons son approche ludique dans le cadre de la programmation afin d'introduire les fondements de la virologie informatique.

Nous invitons le lecteur à se prêter au jeu suivant. Nous donnons une phrase commandant l'écriture d'un groupe de mots et le lecteur devra exécuter cette commande. Prenons un exemple.

Ecrire son prénom. (1.1)

Cette phrase invite le lecteur à écrire son prénom sur la feuille dont il s'est préalablement muni.

Pointeurs. Le terme *auto-référence* désigne le fait qu'un objet puisse se désigner lui-même. La phrase suivante est auto-référente dans le sens où l'action qu'elle exprime fait référence à elle-même en tant qu'objet syntaxique.

Ecrire cette phrase. (1.2)

Pour construire la phrase (1.2) nous avons utilisé le mot « *cette* ». Ce mot est tout à fait particulier puisque qu'il désigne un élément de son contexte. Ici il désigne la phrase (1.2).

Le lecteur familier avec la théorie de la programmation aura reconnu la notion de *pointeur*. Un pointeur exprime un raccourci sémantique. En l'occurrence, le groupe de mots « *cette phrase* » est un pointeur désignant la phrase « *Ecrire cette phrase.* ». Les pointeurs permettent de construire facilement des objets auto-référents mais ce n'est pas le seul moyen.

Sémantique et syntaxe. Nous avons mis en évidence une relation entre la sémantique et la syntaxe des phrases. La frontière entre ces deux notions est parfois ambiguë,

comme l'illustrent les deux phrases suivantes.

Ecrire le vent. (1.3)

Ecrire vent en quatre lettres. (1.4)

La première phrase fait référence à l'objet sémantique désigné par le mot « *vent* ». Ainsi l'action exprimée par cette phrase tient de l'exercice de poésie. La seconde phrase nous invite à écrire le mot « *vent* ». Les mots peuvent être équivoques : suivant leur contexte ils font référence soit à leur sémantique, soit à leur syntaxe.

Pour conserver une certaine lisibilité, il est nécessaire de lever cette ambiguïté. Ainsi, on considère que la phrase (1.4) est mal formée. Dorénavant, pour faire référence à un groupe de mots en tant qu'objet syntaxique, nous le placerons entre guillemets. On préférera la phrase suivante à la phrase (1.4).

Ecrire « *vent* » en quatre lettres.

Diagonalisation. Pour construire une phrase auto-référente sans l'emploi de pointeur, on utilise généralement une *diagonalisation*. Ce mot barbare a été introduit par Gödel [Göd31] afin de nommer la méthode qu'il utilisa pour montrer son célèbre théorème d'incomplétude.

En pratique, on se munit d'un symbole $*$ pouvant être substitué par tout groupe de mots. Par exemple la phrase (1.5) résulte de la substitution du symbole $*$ par le groupe de mots « *le vent* » dans la phrase « Ecrire $*$. ».

Ecrire le vent. (1.5)

Nous définissons la *digonalisation d'une phrase* comme la phrase résultant de la substitution du symbole $*$ par cette première phrase encadrée de guillemets. Par exemple la phrase suivante est la digonalisation de la phrase « *Ecrire* $*$. ».

Ecrire « *Ecrire* $*$. ». (1.6)

Nous avons tous les éléments pour construire une phrase auto-référente. Il suffit de prendre la diagonalisation de « *Ecrire la diagonalisation de* $*$. ».

Ecrire la diagonalisation de « *Ecrire la diagonalisation de* $*$. ». (1.7)

Nous invitons le lecteur à effectuer l'action exprimée par la phrase (1.7). Il observera qu'il est amené à écrire cette même phrase. Nous avons construit une phrase auto-référente sans l'aide d'aucun pointeur.

Normalisation. Il existe de nombreuses méthodes permettant de construire une phrase auto-référente. Nous en présentons une deuxième. On définit la *normalisation*

d'une phrase comme cette phrase suivie d'elle même entre guillemets. Par exemple la phrase suivante est la normalisation de « *Ecrire le vent.* ».

Ecrire le vent « *Ecrire le vent.* ». (1.8)

Le lecteur observera que la phrase (1.9) est auto-référente alors qu'elle ne comporte ni pointeur ni symbole.

Ecrire la normalisation de « *Ecrire la normalisation de.* ». (1.9)

Offuscation. Au vu des constructions précédentes, on pourrait penser qu'il soit possible d'identifier les phrase auto-référentes par le fait qu'elles comportent une répétition. Nous montrons que cette répétition peut être aisément dissimulée.

On définit le codage d'une lettre comme la lettre située treize positions plus loin dans l'alphabet. Si on arrive au bout de l'alphabet, on continue l'opération depuis le début. Ce code est communément appelé Rot 13. Le tableau suivant donne le codage des différentes lettres de l'alphabet.

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m

Comme l'alphabet est composé de vingt six lettres, on observera que le codage du codage d'une lettre donne la lettre originale.

Le codage d'une phrase est la phrases résultant du codage de toutes ses lettres. Par exemple la phrase suivante est le codage de « *Ecrire le vent.* ».

Rpever yr irag. (1.10)

Un codage analogue fut utilisé par l'empereur César pour dissimuler ses communications avec ses généraux. Ici nous l'utilisons pour dissimuler le mécanisme d'auto-référence.

La *normalisation codée d'une phrase* est cette phrase suivie de son codage entre guillemets. Par exemple la phrase suivante est la normalisation codée de « *Ecrire le vent.* ».

Ecrire le vent « *Rpever yr irag.* ». (1.11)

Le lecteur observera que la phrase (1.12) est auto-référente, pourtant elle ne présente pas de répétition.

Ecrire le codage de la normalisation codée de
« *Rpever yr pbqntr qr yn abeznyvfngvba pbqŕr qr.* ». (1.12)

En virologie informatique, on parle d'offuscation lorsqu'un virus emploie de tels moyens de dissimulation.

Virus informatiques. Un programme informatique est une phrase, un ordinateur étant une machine exécutant les actions exprimées par les programmes. Un *virus informatique* est un programme capable de se propager à travers les ordinateurs. En d'autres termes un virus[1] est une phrase telle que les actions qu'elle exprime conduit à sa reproduction. Par exemple le programme suivant peut être considéré comme un virus.

Envoyer cette phrase à tous les ordinateurs. (1.13)

Si cette phrase est exécutée par un ordinateur elle sera envoyée à tous les autres ordinateurs accessibles. Ces derniers exécuteront alors cette même phrase et ainsi de suite. Très vite tous les ordinateurs ne feront qu'envoyer ce virus à tous les autres ordinateurs. Ce flux risque de saturer les réseaux de communication.

Ce scénario n'est pas anodin, il est très proche de l'infection informatique engendrée par le ver `Sapphire/Slammer` en 2003. En seulement 10 minutes cette infection a compromis plus de 75'000 ordinateurs. Elle causa de nombreux incidents : annulation de vols, interruption de processus d'élections, isolement de la Corée du Sud vis à vis de l'internet.

Les constructions auto-référentes constituent un des piliers de la virologie informatique. Nous avons montré qu'il ne suffit pas de faire en sorte que les programmes ne possèdent pas de pointeur pour résoudre le problème. En effet les phrases suivantes conduisent au même résultat.

Envoyer la diagonalisation de
« *Envoyer la diagonalisation de* $*$ *à tous les ordinateurs.* » (1.14)
à tous les ordinateurs.

Envoyer à tous les ordinateurs la normalisation de
« *Envoyer à tous les ordinateurs la normalisation de.* ». (1.15)

Envoyer à tous les ordinateurs le codage de la normalisation codée de
« *Raiblre ṅ gbhf yrf beqvangrhe yr pbqntr qr yn abeznyvfngvba pbqŕr qr.* ». (1.16)

Les chapitres suivants sont consacrés à l'étude de ces mécanismes. Ce travail nous permettra parfois d'entrevoir des solutions. Néanmoins, son objectif principal reste la vulgarisation des fondements de la virologie informatique afin de promouvoir la recherche dans ce domaine.

[1] Dans ce travail nous n'étudions que des virus informatiques, ainsi nous abrégerons régulièrement ce terme par le mot *virus*.

2 Le langage `While`

Pour étudier les virus informatiques, il est nécessaire de définir une notion robuste de programme. Pour cela, nous recourons à un langage de programmation particulier nommé `While`. Nous empruntons cette vision moderne de la calculabilité à Jones [Jon97]. Une démarche similaire utilisant le langage `1#` est proposée par Moss [Mos06]. Au delà de sa dimension constructive, cette approche a l'avantage d'être pédagogique. En effet, le langage `While` est simple à présenter et à comprendre.

Ce langage prend la forme d'un ensemble de procédures opérant des calculs sur les arbres binaires. Nous commençons par définir une syntaxe abstraite à laquelle est adjointe une sémantique. Ensuite, nous donnons une syntaxe concrète permettant de représenter les programmes à l'intérieur du domaine de calcul. Nous pouvons ainsi construire des transformateurs de programme, c'est à dire des programmes modifiant ou construisant d'autres programmes. Cet aspect est essentiel à la virologie informatique puisqu'un virus peut être vu comme un programme transformant un programme hôte en un programme infecté.

2.1 Environnement de calcul

Domaine de calcul. Le *domaine de calcul* regroupe les objets sur lesquels opèrent les programmes. Nous travaillons sur l'ensemble des arbres binaires construit à partir des symboles $\bullet$ et $\circ$ d'arités respectives 0 et 2. On rappel que l'arité correspond au nombre d'arguments d'un symbole. Le symbole $\bullet$ se prononce *nil* et $\circ$ se prononce *cons*. L'ensemble des arbres binaires $\mathbb{D}$ est récursivement défini par l'équation suivante.

$$\mathbb{D} ::= \bullet \mid \circ(\mathbb{D}, \mathbb{D})$$

La figure 2.1 présente plusieurs exemples d'arbres binaires ainsi que leurs représentations graphiques usuelles. Afin de faciliter la lecture, nous utilisons le plus souvent possible la représentation graphique.

Mémoire. La *mémoire* est un container pouvant accueillir des éléments du domaine de calcul. Nous représentons cette mémoire par un ensemble de variables noté $\mathbb{X}$. Une *assignation* σ associe toute variable $x \in \mathbb{X}$ à un arbre binaire $t \in \mathbb{D}$. En

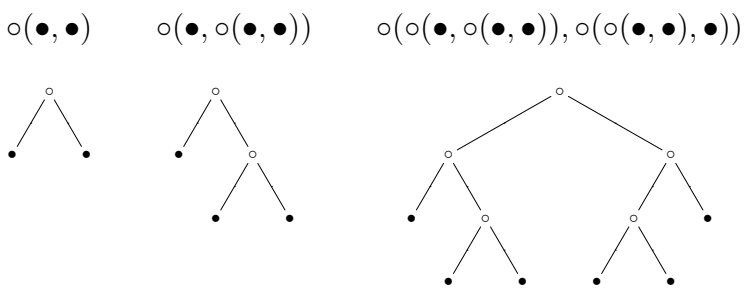


FIG. 2.1: Arbres binaire

d'autres termes, σ est une fonction de $\mathbb{X}$ vers $\mathbb{D}$. On note $\mathbb{S} = (\mathbb{X} \to \mathbb{D})$ l'ensemble des assignations.

Afin de redéfinir les valeurs d'une assignation, on utilise la notation suivante. Pour toute assignation $\sigma \in \mathbb{S}$, toute variable $x \in \mathbb{X}$ et tout arbre $t \in \mathbb{D}$ on a

$$\sigma[x \mapsto t](y) = \begin{cases} t & \text{si } y = x \\ \sigma(y) & \text{sinon} \end{cases}$$

Par commodité nous considérons que $\mathbb{X}$ est un ensemble infini. Nous aurions pu nous restreindre à une unique variable mais le développement aurait été plus fastidieux. Nous invitons tout de même le lecteur à consulter [Jon97] où Jones montre l'équivalence entre le langage `While` à une variable et sa version à un nombre arbitraire de variables.

2.2 Syntaxe et sémantique

Expressions. Une *expression* permet de lire des emplacements de la mémoire, de les comparer, et de construire de nouvelles données. On se munit de deux symboles *hd* et *tl* d'arité 1 et d'un symbole `==` d'arité 2. Ces symboles se prononcent respectivement *head*, *tail* et *compare*. L'ensemble des expressions $\mathbb{E}$ est récursivement défini par

$$\mathbb{E} ::= \mathbb{D} \mid \mathbb{X} \mid \ \circ(\mathbb{E}, \mathbb{E}) \mid \ hd(\mathbb{E}) \mid \ tl(\mathbb{E}) \mid \mathbb{E} \texttt{==} \mathbb{E}$$

Etant donné une assignation $\sigma \in \mathbb{S}$, l'évaluation d'une expression $e \in \mathbb{E}$ retourne un arbre binaire noté $(\!|e|\!)(\sigma)$. Cette évaluation est récursivement définie de la manière suivante.

Constantes. L'évaluation d'une donnée est la donnée elle-même : pour tout arbre binaire $t \in \mathbb{D}$ on a

$$(\!|t|\!)(\sigma) = t$$

Variables. L'évaluation d'une variable est la valeur qui lui est assignée : pour toute variable $x \in \mathbb{X}$ on a

$$(\!|x|\!)(\sigma) = \sigma(x)$$

Couples. L'évaluation d'un couple d'expressions est le couple des évaluations des expressions : pour toutes expressions $e_1, e_2 \in \mathbb{E}$ on a

$$(\!|\circ(e_1, e_2)|\!)(\sigma) = \circ((\!|e_1|\!)(\sigma), (\!|e_2|\!)(\sigma))$$

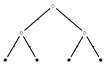
Projections. L'évaluation de *hd* et *tl* retourne respectivement la première et la seconde projection de leur argument : pour toute expression $e \in \mathbb{E}$ on a

$$(\!|hd\,(e)|\!)(\sigma) = \begin{cases} t_1 & \text{si } (\!|e|\!)(\sigma) = \circ(t_1, t_2) \\ \bullet & \text{sinon} \end{cases} \qquad (\!|tl\,(e)|\!)(\sigma) = \begin{cases} t_2 & \text{si } (\!|e|\!)(\sigma) = \circ(t_1, t_2) \\ \bullet & \text{sinon} \end{cases}$$

Comparaisons. L'évaluation d'une comparaison `==` retourne $\circ(\bullet, \bullet)$ en cas d'égalité de l'évaluation de ses arguments et elle retourne $\bullet$ sinon : pour toutes expressions $e_1, e_2 \in \mathbb{E}$ on a

$$(\!|e_1 \texttt{==} e_2|\!)(\sigma) = \begin{cases} \circ(\bullet, \bullet) & \text{si } (\!|e_1|\!)(\sigma) = (\!|e_2|\!)(\sigma) \\ \bullet & \text{sinon} \end{cases}$$

Nous donnons quelques exemples d'évaluations d'expressions. Soient $x \in \mathbb{X}$ une variable et $\sigma \in \mathbb{S}$ une l'assignation telles que $\sigma(x) =$  . On a les évaluations suivantes.

$(\!|\circ(\bullet, x)|\!)(\sigma) =$  $\qquad (\!|\circ(tl\,(x), tl\,(x))|\!)(\sigma) =$ 

$(\!|hd\,(x)|\!)(\sigma) = \bullet \qquad (\!|tl\,(x)|\!)(\sigma) =$ 

$\left(\!\left|x \texttt{==} \right.\right.$  $\left.\left|\right)\right.(\sigma) =$  $\qquad (\!|x \texttt{==} tl\,(x)|\!)(\sigma) = \bullet$

Commandes. On se munit des symboles `=`, `;` et *while* d'arité 2 respectivement prononcés *assigne*, *compose* et *while*. L'ensemble des commandes du langage `While` est récursivement défini par l'équation suivante.

$$\mathbb{C} ::= \mathbb{X} \texttt{=} \mathbb{E} \mid \mathbb{C}\,\texttt{;}\,\mathbb{C} \mid \mathit{while}\,(\mathbb{E})\{\mathbb{C}\}$$

Une *commande* est une opération sur la mémoire : l'exécution d'une commande prend une assignation et retourne une nouvelle assignation. Plus formellement, l'exécution d'une commande $c \in \mathbb{C}$ sur une assignation $\sigma \in \mathbb{S}$ retourne une assignation noté $[\![c]\!](\sigma)$ définie comme suit.

Assignation. Pour toute variable $x \in \mathbb{X}$ et toute expression $e \in \mathbb{E}$ l'exécution de $x = e$ réassigne la variable x à la valeur de l'expression e dans σ.

$$[\![x = e]\!](\sigma) = \sigma[x \mapsto (\!|e|\!)(\sigma)]$$

Composition. Pour toutes commandes $c_1, c_2 \in \mathbb{C}$ l'exécution de $c_1 \,;\, c_2$ correspond à l'exécution successive des commandes c_1 et c_2.

$$[\![c_1 \,;\, c_2]\!](\sigma) = [\![c_2]\!]([\![c_1]\!](\sigma))$$

Boucles. Pour toute expression $e \in \mathbb{E}$ et toute commande $c \in \mathbb{C}$ l'exécution de $\mathit{while}\,(e)\{c\}$ correspond à exécuter la commande c tant que l'évaluation de l'expression e est différente de $\bullet$.

$$[\![\mathit{while}\,(e)\{c\}]\!](\sigma) = \begin{cases} \sigma & \text{si } (\!|e|\!)(\sigma) = \bullet \\ [\![c \,;\, \mathit{while}\,(e)\{c\}]\!](\sigma) & \text{sinon} \end{cases}$$

Nous donnons quelques exemples d'exécutions de commandes. Soient $x, y \in \mathbb{X}$ des variables et $\sigma \in \mathbb{S}$ une assignation telles que $\sigma(x) =$  et $\sigma(y) =$  . Nous avons les exécutions suivantes

$$[\![y = \circ\,(\bullet, y)]\!](\sigma) = \sigma' \text{ avec } \sigma'(x) = \text{  } \quad \text{et } \sigma'(y) = \text{  }$$

$$[\![y = \circ\,(\bullet, y) \,;\, x = \mathit{tl}\,(x)]\!](\sigma) = \sigma' \text{ avec } \sigma'(x) = \text{} \quad \text{et } \sigma'(y) = \text{  }$$

$$[\![\mathit{while}\,(x)\{y = \circ\,(\bullet, y) \,;\, x = \mathit{tl}\,(x)\}]\!](\sigma) = \sigma' \text{ avec } \sigma'(x) = \bullet \quad \text{et } \sigma'(y) = \text{  }$$

On remarquera que la sémantique de *while* implique que l'exécution de certaines commandes n'est pas définie. Par exemple l'exécution de $\mathit{while}\,(\circ(\bullet, \bullet))\{x = x\}$ ne termine pas.

Programmes. On se munit d'une variable particulière $\mathit{work} \in \mathbb{X}$ et on note $[\mathit{work} \mapsto t]$ l'assignation définie pour toute variable $x \in \mathbb{X}$ par

$$[\mathit{work} \mapsto t](x) = \begin{cases} t & \text{si } x = \mathit{work} \\ \bullet & \text{sinon} \end{cases}$$

Un *programme* de `While` est simplement une commande. En notant $\mathbb{P}$ l'ensemble des programmes on a $\mathbb{P} = \mathbb{C}$. A la différence d'une commande, l'exécution d'un programme $\mathbf{P} \in \mathbb{P}$ s'effectue sur une entrée t. Elle consiste à exécuter $\mathbf{P}$ en tant que commande sur l'assignation $[\mathit{work} \mapsto t]$. Si cette exécution termine, le programme retourne le nouvel arbre binaire assignée à la variable work. On a pour tout programme $\mathbf{P} \in \mathbb{P}$ et toute donnée $t \in \mathbb{D}$

$$[\![\mathbf{P}]\!](t) = \sigma(\mathit{work}) \text{ avec } \sigma = [\![\mathbf{P}]\!]([\mathit{work} \mapsto t])$$

$$
\begin{array}{rl}
\text{Domaine :} & \mathbb{D} ::= \bullet \mid \circ(\mathbb{D}, \mathbb{D}) \\
\text{Variables :} & \mathbb{X} ::= work \mid x \mid y \mid \cdots \\
\text{Expressions :} & \mathbb{E} ::= \mathbb{D} \mid \mathbb{X} \mid \circ(\mathbb{E}, \mathbb{E}) \mid hd(\mathbb{E}) \mid tl(\mathbb{E}) \mid \mathbb{E} == \mathbb{E} \\
\text{Commandes :} & \mathbb{C} ::= \mathbb{X} = \mathbb{E} \mid \mathbb{C} ; \mathbb{C} \mid while(\mathbb{E})\{\mathbb{C}\} \\
\text{Programmes :} & \mathbb{P} ::= \mathbb{C}
\end{array}
$$

FIG. 2.2: Syntaxe du langage `While`

$$\forall \sigma \in \mathbb{S}, t \in \mathbb{D}, x \in \mathbb{X}, e, e_1, e_2 \in \mathbb{E}, c, c_1, c_2 \in \mathbb{C}, \mathbf{p} \in \mathbb{P}$$

$$
\begin{array}{rl}
\text{Domaine :} & (\!|t|\!)(\sigma) = t \\
\text{Variables :} & (\!|x|\!)(\sigma) = \sigma(x) \\
\text{Expressions :} & (\!|\circ(e_1, e_2)|\!)(\sigma) = \circ((\!|e_1|\!)(\sigma), (\!|e_2|\!)(\sigma)) \\
& (\!|hd(e)|\!)(\sigma) = \begin{cases} \bullet & \text{si } (\!|e|\!)(\sigma) = \bullet \\ t_1 & \text{si } (\!|e|\!)(\sigma) = \circ(t_1, t_2) \end{cases} \\
& (\!|tl(e)|\!)(\sigma) = \begin{cases} \bullet & \text{si } (\!|e|\!)(\sigma) = \bullet \\ t_2 & \text{si } (\!|e|\!)(\sigma) = \circ(t_1, t_2) \end{cases} \\
& (\!|e_1 == e_2|\!)(\sigma) = \begin{cases} \circ(\bullet, \bullet) & \text{si } (\!|e_1|\!)(\sigma) = (\!|e_2|\!)(\sigma) \\ \bullet & \text{sinon} \end{cases} \\
\text{Commandes :} & [\![x = e]\!](\sigma) = \sigma[x \mapsto (\!|e|\!)(\sigma)] \\
& [\![c_1 ; c_2]\!](\sigma) = [\![c_2]\!]([\![c_1]\!](\sigma)) \\
& [\![while(e)\{c\}]\!](\sigma) = \begin{cases} \sigma & \text{si } (\!|e|\!)(\sigma) = \bullet \\ [\![while(e)\{c\}]\!]([\![c]\!](\sigma)) & \text{sinon} \end{cases} \\
\text{Programmes :} & [\![\mathbf{P}]\!](t) = \sigma(work) \text{ avec } \sigma = [\![\mathbf{P}]\!]([work \mapsto t])
\end{array}
$$

FIG. 2.3: Sémantique du langage `While`

Syntaxe et sémantique en résumé. Nous résumons la syntaxe et la sémantique du langage `While` par les figures 2.2 et 2.3.

2.3 Raccourcis syntaxiques et macros

Expressions. Afin de pouvoir écrire des programmes de manière concise et lisible, nous utiliserons les abréviations suivantes.

- On note $\circ(e_1, \cdots, e_n) = \circ(e_1, \cdots, \circ(e_{n-1}, e_n)\cdots)$ et $\circ(e) = e$.
- On utilise $e_1 \,!\!=\, e_2$ pour désigner l'expression $(e_1 == e_2) == \bullet$. On observe que cette notation permet de tester si deux expressions ont des évaluations distinctes.
- L'abréviation $hd^n\,(e)$ (resp. $tl^n\,(e)$) désigne l'application de n symboles hd (resp. tl) successifs à l'expression e.

Commandes De même nous ajoutons les commandes suivantes.

- Pour toutes variables $x_1, \cdots, x_n \in \mathbb{D}$ la notation $[x_1, \cdots, x_n] = work$ désigne la commande suivante.

$$\begin{array}{l} x_1 = hd\,(work)\ ; \\ \cdots \\ x_{n-1} = hd\,(tl^{n-2}\,(work))\ ; \\ x_n = tl^{n-1}\,(work) \end{array}$$

  Cette commande nous permet de décomposer l'entrée d'un programme en n paramètres assignés aux variables $x_1, \cdots, x_n$. En effet, on observe que l'évaluation de la commande $[x_1, \cdots, x_n] = work$ sur l'assignation $[work \mapsto \circ(t_1, \cdots, x_n)]$ retourne une assignation σ telle que pour tout $m \in \{1, \cdots, n\}$ on a $\sigma(x_m) = t_m$.
- La notation $skip$ désigne la commande $work = work$. On observe que cette commande n'a aucun effet.
- Pour toute expression $e \in \mathbb{E}$ et toutes commandes $c_1, c_2 \in \mathbb{C}$ la notation $if\,(e)\{c_1\}\,else\,\{c_2\}$ désigne la commande suivante où $x_1, x_2 \in \mathbb{X}$ sont deux nouvelle variables n'apparaissant pas dans le reste du programme.

$$\begin{array}{ll} x_1 = e\ ;\ x_2 = e\ ; & \text{// stocke la valeur de } e \text{ pour les deux tests} \\ while\,(x_1 \,!\!=\, \bullet)\{ & \text{// si la valeur de } e \text{ est différente de } \bullet \\ \quad c_1\ ; & \text{// évalue la commande } c_1 \\ \quad x_1 = \circ\,(\bullet, \bullet)\ ; & \text{// fait en sorte de sortir de la boucle} \\ \}\ ; & \\ while\,(x_2 == \bullet)\{ & \text{// si la valeur de } e \text{ est égale à } \bullet \\ \quad c_2\ ; & \text{// évalue la commande } c_1 \\ \quad x_2 = \bullet\ ; & \text{// fait en sorte de sortir de la boucle} \\ \} & \end{array}$$

On observe que si l'évaluation de l'expression e est différente de $\bullet$ alors la commande c_1 est exécutée, sinon la commande c_2 est exécutée.

– Pour toute expression $e \in \mathbb{E}$ et toute commande $c \in \mathbb{C}$ la notation $if\,(e)\{c\}$ désigne la commande $if\,(e)\{c\}\,else\,\{skip\}$. On observe que si l'évaluation de l'expression e est différente de $\bullet$ alors la commande c_1 est exécutée, sinon rien ne se passe.
– Pour toute variable $x \in \mathbb{X}$, toute expression $e \in \mathbb{E}$ et toute commande $c \in \mathbb{C}$ la notation $foreach\,(x\ in\ e)$ désigne la commande suivante où $y \in \mathbb{X}$ est une nouvelle variable.

```
y = e ;          // évalue l'expression e
while (y){       // tant que y est différente de •
 x = hd (y)      // assigne la tête de y à x
 c ;             // exécute la commande c
 y = tl (y)      // passe à la valeur suivante
}
```

Dans cette commande la valeur de l'expression e est vue comme une liste de la forme $\circ(t_1, \cdots, t_n, \bullet)$. Ainsi, on exécute n fois la commande c où x prend successivement les valeurs $t_1, \cdots, t_n$.

Programmes. Nous introduisons un mécanisme de *macro*. Pour tout programme $\mathbf{P} \in \mathbb{P}$, toute variable $x \in \mathbb{X}$ et toutes expressions $e_1, \cdots, e_n \in \mathbb{E}$, la notation $x = \mathbf{P}(e_1, \cdots, e_n)$ désigne la commande suivante ou $y \in \mathbb{X}$ est une nouvelle variable.

```
y = work ;                  // sauvegarde la valeur de work
work = ∘ (e_1, ··· , e_n) ; // construit l'entrée pour le programme P
P ;                         // exécute P
x = work ;                  // assigne la sortie à la variable x
work = y ;                  // restaure la valeur de work
```

On rappelle que nous disposons d'un nombre arbitraire de variables. Ainsi à un renommage près, on peut considérer que les variables apparaissant dans les macros sont distinctes des variables. En d'autres termes, on supposera qu'il n'y a aucun problème de collision de variables.

2.4 Syntaxe concrète.

La *syntaxe concrète* de `While` permet de représenter les programmes de $\mathbb{P}$ à l'intérieur du domaine de calcul $\mathbb{D}$.

On se munit d'une application injective de $\mathbb{X}$ vers $\mathbb{D}$, pour toute variable $x \in \mathbb{X}$ on notera $\overline{x}$ l'image de x par cette application. En d'autres termes, l'application $x \mapsto \overline{x}$ est une représentation de l'ensemble des variables dans le domaine de calcul.

Soient $\mathit{quote}, \mathit{var}, \circ, \mathit{hd}, \mathit{tl}, \texttt{==}, \texttt{=}, \texttt{;}, \mathit{while} \in \mathbb{D}$ des arbres binaires distincts. On définit l'application qui à tout programme $\mathbf{P} \in \mathbb{P}$ associe un arbre binaire noté $\underline{\mathbf{P}}$ par

$$
\begin{aligned}
\text{Domaine :} \quad & \underline{t} = \circ(\mathit{quote}, t) \\
\text{Variables :} \quad & \underline{x} = \circ(\mathit{var}, \bar{x}) \\
\text{Expressions :} \quad & \underline{\circ(e_1, e_2)} = \circ(\circ, \underline{e_1}, \underline{e_2}) \\
& \underline{hd\,(e_1)} = \circ(\mathit{hd}, \underline{e_1}) \\
& \underline{tl\,(e_1)} = \circ(\mathit{tl}, \underline{e_1}) \\
& \underline{e_1 \texttt{ == } e_2} = \circ(\texttt{==}, \underline{e_1}) \\
\text{Commandes :} \quad & \underline{x \texttt{ = } e_1} = \circ(\texttt{=}, \underline{x}, \underline{e_1}) \\
& \underline{c_1 \texttt{ ; } c_2} = \circ(\texttt{;}, \underline{c_1}, \underline{c_2}) \\
& \underline{while\,(e)\{c\}} = \circ(\mathit{while}, \underline{e}, \underline{c})
\end{aligned}
$$

On étend la notation $[\![\ ]\!]$ sur $\mathbb{D}$ de la manière suivante. Pour tout arbre binaire $\mathbf{p} \in \mathbb{D}$, s'il existe un programme $\mathbf{P} \in \mathbb{P}$ tel que $\mathbf{p} = \underline{\mathbf{P}}$ alors $[\![\mathbf{p}]\!] = [\![\mathbf{P}]\!]$ sinon $[\![\mathbf{p}]\!]$ est la fonction constante égale à $\bullet$, c'est à dire que pour tout arbre $t \in \mathbb{D}$ on a $[\![\mathbf{p}]\!]\,(t) = \bullet$. Ainsi $[\![\ ]\!]$ est une fonction de $\mathbb{D} \to \mathbb{D} \to \mathbb{D}$.

Par la suite, nous confondrons les programmes de $\mathbb{P}$ et leur syntaxe concrète sur $\mathbb{D}$. Comme ces deux éléments représentent le même objet, cette liberté ne devrait pas poser de problème au lecteur.

2.5 Temps d'exécution

Temps d'exécution. Le *temps d'exécution* d'un programme correspond au nombre d'étapes élémentaires nécessaires à l'exécution de ce programme. Le temps d'exécution des programmes du langage `While` est donnée par la fonction T définie comme suit. Pour tout $\sigma \in \mathbb{S}, t \in \mathbb{D}, x \in \mathbb{X}, e, e_1, e_2 \in \mathbb{E}, c, c_1, c_2 \in \mathbb{C}, \mathbf{p} \in \mathbb{P}$

$$
\begin{aligned}
\text{Domaine :} \quad & T_t(\sigma) = 1 \\
\text{Variables :} \quad & T_{x_1}(\sigma) = 1 \\
\text{Expression :} \quad & T_{\circ(e_1,e_2)}(\sigma) = 1 + T_{e_1}(\sigma) + T_{e_2}(\sigma) \\
& T_{hd(e_1)}(\sigma) = 1 + T_{e_1}(\sigma) \\
& T_{tl(e_1)}(\sigma) = 1 + T_{e_1}(\sigma) \\
& T_{e_1 \texttt{==} e_2}(\sigma) = 1 + T_{e_1}(\sigma) + T_{e_2}(\sigma) \\
\text{Commandes :} \quad & T_{x \texttt{=} e_1}(\sigma) = 1 + T_{e_1}(\sigma) \\
& T_{c_1 \texttt{;} c_2}(\sigma) = T_{c_1}(\sigma) + T_{c_2}([\![c_1]\!]\,(\sigma)) \\
& T_{while(e)\{c\}}(\sigma) = \begin{cases} T_e(\sigma) & \text{si } (\!|e|\!)\,(\sigma) = \bullet \\ T_e(\sigma) + T_c(\sigma) + T_{while(e)\{c\}}([\![c]\!]\,(\sigma)) & \text{sinon} \end{cases} \\
\text{Programmes :} \quad & T_{\mathbf{P}}(t) = T_{\mathbf{P}}([work \mapsto t])
\end{aligned}
$$

3 Eléments de calculabilité

Pour définir la notion de programme, nous avons utilisé un unique langage de programmation. Nous justifions cette approche en nous reposant sur le théorème d'isomorphisme de Rogers. Il assure que le langage `While` est isomorphe à tout langage de programmation acceptable. Il est communément admis que la notion d'acceptabilité permet de caractériser la majorité des modèles de calcul raisonnables. Ainsi l'étude du langage `While` permet de de rendre compte des propriétés communes à la majorité des langages de programmation usuels.

Par conséquent, le langage `While` possède deux avantages majeurs. D'une part, sa simplicité permet d'exhiber des exemples concrets. L'exposition est ainsi plus pertinente. D'autre part, le fait que ce langage soit acceptable nous garantit une certaine généralité des résultats : à un isomorphisme près, nos constructions sont reproductibles dans la majorité des modèles de calcul.

Nous commençons par définir la notion d'acceptabilité d'un langage de programmation puis nous présentons le théorème d'isomorphisme de Rogers. Au long de ce développement nous rencontrerons les notions d'auto-interprétation et de spécialisation, deux objets fondamentaux pour la suite.

Dans un deuxième temps, nous exposons l'un des théorèmes les plus profonds de l'informatique fondamentale : le second théorème de récursion de Kleene. Ce théorème fournit une construction systématique de programmes auto-référents. Nous montrons que les programmes produits sont relativement efficaces. Enfin, nous présentons quelques applications. Nous nous attardons sur ce théorème car il est au cœur de notre définition de la notion de virus informatique.

3.1 Langages de programmation

Domaine de calcul. Nous rappelons que le *domaine de calcul* est l'ensemble des objets sur lesquels opèrent les programmes. Nous avons choisi de travailler sur les arbres binaires car se sont des objets faciles à manipuler et à représenter. Toutefois ce choix reste arbitraire, nous aurions pu y préférer les entiers naturels ou un ensemble de mots construits sur un alphabet fixé. Du point de vue de la calculabilité [1], chacune de ces représentations pouvant être encodée dans toutes les autres, elles sont équivalentes.

[1] En complexité la représentation des données aurait plus d'importance, par exemple les codages unaire et binaire des entiers n'engendre pas les mêmes classes de complexité.

Langage de programmation. Un *langage de programmation* est une application $\varphi : \mathbb{D} \to (\mathbb{D} \to \mathbb{D})$ qui à tout élément du domaine de calcul associe une fonction de $\mathbb{D} \to \mathbb{D}$. Pour tout élément $\mathbf{p} \in \mathbb{D}$, on note $\varphi_{\mathbf{p}}$ la fonction associée à $\mathbf{p}$ et on dit que $\varphi_{\mathbf{p}}$ est la *fonction calculée* par le *programme* $\mathbf{p}$. De plus, pour tout élément $t \in \mathbb{D}$ on dira que $\varphi_{\mathbf{p}}(t)$ est la *sortie* du programme $\mathbf{p}$ exécuté sur l'*entrée* t.

Une fonction $\mathcal{F}$ est *totale* si pour tout élément $t \in \mathbb{D}$ la valeur $\mathcal{F}(t)$ est définie. Une fonction $\mathcal{F}$ est dite *φ-semi-calculable* s'il existe un programme $\mathbf{p} \in \mathbb{D}$ tel que $\mathcal{F} = \varphi_{\mathbf{p}}$. Si de plus $\mathcal{F}$ est totale, on dit que $\mathcal{F}$ est *φ-calculable*.

Les *données* sont les objets apparaissant comme entrée ou sortie d'un programme. Nous constatons que ce formalisme ne fait aucune distinction entre données et programmes. Ce point de vue est important pour la modélisation des virus informatiques. Intuitivement, un virus est un programme prenant comme entrée un programme hôte et retournant en sortie une version infectée de cet hôte. Ainsi, on observe que les données peuvent être des programmes.

Cette approche n'est pas propre à la virologie informatique. Elle se retrouve dans d'autres champs de l'informatique comme celui de la théorie de la compilation. En effet, un compilateur n'est rien d'autre qu'un programme prenant en entrée un programme écrit dans un langage source et retournant en sortie un programme écrit dans un langage cible.

Néanmoins, par soucis de lisibilité, nous distinguons dans la mesure du possible les éléments du domaine vus comme des programmes et ceux vus comme des données. Cette dichotomie prendra la forme de deux notations, les symboles $\mathbf{p}, \mathbf{q}, \cdots$ désignent des programmes et les symboles $t, u, \cdots$ désignent des données.

Couplage. Une *fonction de couplage* est une fonction surjective de $\mathbb{D} \times \mathbb{D}$ vers $\mathbb{D}$. Une telle fonction permet de décomposer les données en deux sous-données nommées *arguments*.

Par abus de notation on écrit $\langle t_1, \cdots, t_{n-1}, t_n \rangle = \langle t_1, \cdots, \langle t_{n-1}, t_n \rangle \cdots \rangle$ et par convention $\langle t \rangle = t$. Cette extension permet de décomposer les données en un nombre arbitraire d'arguments. Ainsi on peut considérer les fonctions d'arité multiple par la convention suivante.

$$\mathcal{F}(t_1, \cdots, t_n) = \mathcal{F}(\langle t_1, \cdots, t_n \rangle)$$

Nous prenons comme fonction de couplage la fonction $\langle t_1, t_2 \rangle = \circ(t_1, t_2)$. Toujours par abus de notation $\circ(t_1, \cdots, t_n) = \circ(t_1, \cdots \circ (t_{n-1}, t_n) \cdots)$. Cet arbre est représenté par la figure 3.1.

Auto-interprétation. La *fonction universelle* $\mathit{univ}_\varphi : \mathbb{D} \to \mathbb{D}$ d'un langage de programmation φ est définie pour tout programme $\mathbf{p} \in \mathbb{D}$ et toute donnée $t \in \mathbb{D}$ par $\mathit{univ}_\varphi(\mathbf{p}, t) = \varphi_{\mathbf{p}}(t)$. Un *auto-interpréteur* d'un langage de programmation φ est

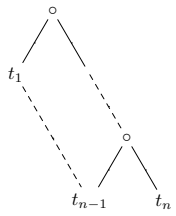


FIG. 3.1: Fonction de couplage, $\circ(t_1, \cdots \circ (t_{n-1}, t_n) \cdots)$

un programme $\mathbf{u} \in \mathbb{D}$ calculant la fonction universelle univ_φ.

$$\varphi_{\mathbf{u}}(\mathbf{p}, t) = \mathit{univ}_\varphi(\mathbf{p}, t) = \varphi_{\mathbf{p}}(t)$$

Un auto-interpréteur est un programme tel que, si on lui fournit pour entrée le couple $\circ(\mathbf{p}, t)$, la sortie obtenue est identique à celle du programme $\mathbf{p}$ exécuté sur l'entrée t.

Spécialisation. Une fonction $\mathit{spec} : \mathbb{D} \to \mathbb{D}$ est une *fonction de spécialisation* si pour tout programme $\mathbf{p} \in \mathbb{D}$ et toute donnée $t \in \mathbb{D}$ le programme $\mathbf{p}_t = \mathit{spec}(\mathbf{p}, t)$ satisfait pour toute donnée $u \in \mathbb{D}$

$$\varphi_{\mathbf{p}_t}(u) = \varphi_{\mathbf{p}}(t, u)$$

A partir d'un programme $\mathbf{p}$ et d'un premier argument t, une fonction de spécialisation construit un nouveau programme $\mathbf{p}_t$ tel que si on l'exécute sur une entrée u, on obtient une sortie identique à celle obtenue par l'exécution de $\mathbf{p}$ sur l'entrée couplé $\circ(t, u)$. En ce sens, on peut voir le programme $\mathbf{p}_t$ comme une spécialisation du programme $\mathbf{p}$ pour l'ensemble restreint d'entrée $\{\circ(t, u) \mid u \in \mathbb{D}\}$.

Une fonction de spécialisation permet aussi de réaliser de l'évaluation partielle, le programme $\mathbf{p}_t = \mathit{spec}(\mathbf{p}, t)$ étant vue comme l'évaluation partielle du programme $\mathbf{p}$ pour la partie fixée de son entrée t. L'évaluation partielle a de nombreuses applications dans le domaine de la compilation et de l'optimisation. Pour plus de détails on se référera à [JGS93].

On remarque que les fonctions de spécialisation sont proches des fonctions Smn définies par Kleene [Kle52]. On rappelle que s^n_m est une fonction Smn si pour tout programme $\mathbf{p} \in \mathbb{D}$ et toutes données $t_1, \cdots, t_m \in \mathbb{D}$ le programme $\mathbf{q} = s^n_m(\mathbf{p}, t_1, \cdots, t_n)$ satisfait pour toutes données $u_1, \cdots, u_m \in \mathbb{D}$

$$\varphi_{\mathbf{q}}(u_1, \cdots, u_n) = \varphi_{\mathbf{p}}(t_1, \cdots, t_m, u_1, \cdots, u_n)$$

Dans notre formalisme, on observe que pour toute fonction de spécialisation spec, la fonction spec_m définie comme suit est une fonction Smn.

$$\mathit{spec}_m(\mathbf{p}, t_1, \cdots, t_m) = \mathit{spec}(\cdots \mathit{spec}(\mathbf{p}, t_1), \cdots, t_m)$$

En effet, en posant $\mathbf{p}_1 = \mathit{spec}(\mathbf{p}, t_1), \cdots, \mathbf{p}_m = \mathit{spec}(\mathbf{p}_{m-1}, t_m)$ on observe d'une part que $\mathbf{p}_m = \mathit{spec}_m(\mathbf{p}, t_1, \cdots, t_m)$ et d'autre part que

$$\varphi_{\mathbf{p}_m}(u_1, \cdots, u_n) = \varphi_{\mathbf{p}_{m-1}}(t_m, u_1, \cdots, u_n) = \cdots = \varphi_{\mathbf{p}}(t_1, \cdots, t_m, u_1, \cdots, u_n)$$

3.2 Acceptabilité

La notion de langage de programmation acceptable a été définie par Uspenskii [Usp56] et Rogers [Rog67]. Elle permet de caractériser des modèles de calcul raisonnables indépendamment de toute machinerie. La majorité des modèles de référence tombent sous l'égide des langages de programmation acceptables. On peut citer les machines de Turing, les fonctions récursives ou le lambda calcul.

Calcul effectif. La notion de fonction *effectivement semi-calculable* est liée à la notion intuitive d'algorithme. Par conséquent, il est difficile d'en donner une définition rigoureuse. Néanmoins, il existe une caractérisation formelle : la célèbre thèse de Church-Turing. Elle identifie les fonctions effectivement semi-calculables aux fonctions calculées par les machines de Turing. Une fonction totale effectivement semi-calculable est dite *effectivement calculable*.

Langage de programmation acceptable. Un langage de programmation φ est *acceptable* s'il satisfait les trois propriétés suivantes.

Turing complet. Une fonction est φ-semi-calculable si et seulement si elle est effectivement semi-calculable.

Universalité. La fonction universelle univ_φ est effectivement semi-calculable.

Spécialisation. Il existe une fonction de spécialisation de φ effectivement calculable.

La propriété de Turing complétude assure que les capacités de calcul du langage φ sont en accord avec la thèse de Church-Turing. D'une part φ doit être un modèle de calcul aussi puissant que les machines de Turing et d'autre part que les capacités de calcul de φ ne doivent pas dépasser celles de ces mêmes machines.

En combinaison avec la propriété de Turing complétude, la propriété d'universalité assure qu'il existe un auto-interpréteur du langage φ. En effet, puisque la fonction univ_φ est effectivement semi-calculable, il existe un programme $\mathbf{u} \in \mathbb{D}$ tel que $\varphi_{\mathbf{u}}(\mathbf{p}, t) = \mathit{univ}(\mathbf{p}, t)$.

La propriété de spécialisation assure simplement qu'il existe un programme calculant une fonction de spécialisation de φ. On appellera un tel programme un *spécialiseur*.

Isomorphisme de Rogers. Le théorème suivant assure que l'on peut rendre compte de l'ensemble des langages de programmation acceptables par un seul d'entre eux.

Théorème 1 (Théorème d'isomorphisme de Rogers)**.** *Pour toute paire de langages de programmation acceptables* $\{\varphi, \psi\}$*, il existe une bijection* $\mathcal{F} : \mathbb{D} \to \mathbb{D}$ *effectivement calculable telle que pour tout programme* $\mathbf{p} \in \mathbb{D}$ *on ait* $\psi_{\mathcal{F}(\mathbf{p})} = \varphi_{\mathbf{p}}$*.*

La démonstration de ce théorème est très intéressante pour les spécialistes. Elle nécessite l'emploi du théorème de récursion de Kleene, que nous présenterons plus loin, et la construction de Cantor-Bernstein. Néanmoins cette démonstration est longue, de plus elle n'est pas nécessaire au développement de cette étude. Ainsi, nous ne démontrons pas ce théorème mais nous renvoyons le lecteur vers [Rog58, Rog67, Jon97].

Le théorème d'isomorphisme de Rogers nous assure que les langages de programmation acceptables sont isomorphes deux à deux. Ainsi, en étudiant l'un d'entre eux on rend compte, à un isomorphisme près, de l'ensemble des langages de programmation acceptables. Cette propriété justifie notre démarche : nous considérons un langage de programmation acceptable nommé `While` et, en nous reposant sur le théorème précédent, nous assurons que les résultats énoncés sont valables pour l'ensemble des langages de programmation acceptables. Ainsi nous couvrons la majorité des modèles de calcul usuels.

Acceptabilité de `While` Dans [Jon97] il est montré que

- Les fonctions calculées par les programmes de `While` sont exactement les fonctions effectivement semi-calculables.
- La fonction universelle de `While` est effectivement semi-calculable.
- `While` admet une fonction de spécialisation effectivement calculable.

On conclut que `While` est un langage de programmation acceptable. Nous ne reproduisons pas les démonstrations des deux premières propriétés. Nous donnons tout de même un spécialiseur de ce langage. La construction suivante sera régulièrement rencontrée dans les chapitres suivants.

On note **spec** le programme suivant.

$$work = \circ\,(;, \circ(=, \underline{work}, \textbf{\textit{quote}}, tl\,(work)), hd\,(work))\;; \qquad (\textbf{spec})$$

Soient $t \in \mathbb{D}$ une donnée et $\mathbf{P} \in \mathbb{P}$ un programme. On observe que si on exécute de **spec** sur l'entrée $\circ(\underline{\mathbf{P}}, t)$ alors l'expression $\circ(=, \underline{work}, \textbf{\textit{quote}}, tl\,(work))$ a pour valeur la syntaxe concrète de la commande $work = \circ\,(t, work)$; et $hd\,(work)$ a pour valeur $\underline{\mathbf{P}}$. Par conséquent $[\![\mathbf{spec}]\!]\,(\underline{\mathbf{P}}, t)$ retourne la syntaxe concrète du programme suivant noté $\mathbf{P}_t$.

$$work = \circ\,(t, work)\;;\mathbf{P} \qquad (\mathbf{P}_t)$$

Soit $u \in \mathbb{D}$ une donnée, on a

$$\begin{aligned}
[\![\mathbf{P}_t]\!]\,(u) &= [\![work = \circ\,(t, work)\,;\mathbf{P}]\!]\,(u) \\
&= [\![work = \circ\,(t, work)\,;\mathbf{P}]\!]\,([work \mapsto u])\,(work) \\
&= [\![\mathbf{P}]\!]\,([work \mapsto \circ(t, u)])\,(work) \\
&= [\![\mathbf{P}]\!]\,(\circ(t, u))
\end{aligned}$$

On conclut que $[\![\mathbf{spec}]\!]$ est une fonction de spécialisation de `While`.

Dans les chapitres suivants **univ**, **spec** et *spec* désignent respectivement un auto-interpréteur de `While`, le spécialiseur que nous venons de construire et la fonction de spécialisation $[\![\mathbf{spec}]\!]$.

3.3 Compilation de points fixes

Nous passons maintenant à l'exposition du second théorème de récursion de Kleene, ou plus simplement nommé théorème de récursion. Il est communément admis [Smu94, Odi89] qu'un modèle de calcul inclus des constructions auto-référentes à partir du moment où il satisfait le théorème de récursion. Comme ce type de construction est l'un des piliers de la virologie informatique, ce théorème est un élément fondamental dans cette étude.

Construction de points fixes. Intuitivement, ce théorème assure que pour toutes fonctions semi-calculables, il existe un programme calculant cette fonction tout en ayant à sa disposition son propre code. Ce programme est nommé *point fixe*.

Théorème 2 (Second théorème de récursion de Kleene)**.** *Pour tout programme* $\mathbf{p} \in \mathbb{D}$ *il existe un programme* $\mathbf{e} \in \mathbb{D}$ *tel que pour toute donnée* $t \in \mathbb{D}$ *on a*

$$[\![\mathbf{e}]\!]\,(t) = [\![\mathbf{p}]\!]\,(\mathbf{e}, t)$$

Démonstration. On pose $\mathbf{k}$ le programme suivant.

$$\begin{array}{l}
x = tl\,(work)\,; \\
work = \mathbf{spec}(hd\,(work), hd\,(work))\,; \\
work = \circ\,(work\,, x)\,; \\
x = \bullet
\end{array} \qquad (\mathbf{k})$$

On observe que pour tout programme $\mathbf{r} \in \mathbb{D}$ et toute donnée $t \in \mathbb{D}$ le programme $\mathbf{q} = \mathbf{k}\,;\mathbf{p}$ satisfait $[\![\mathbf{q}]\!]\,(\mathbf{r}, t) = [\![\mathbf{p}]\!]\,\big(\mathit{spec}(\mathbf{r}, \mathbf{r}), t\big)$. On pose $\mathbf{e} = \mathit{spec}(\mathbf{q}, \mathbf{q})$ et il suit

$$\begin{aligned}
[\![\mathbf{e}]\!]\,(t) &= [\![\mathit{spec}(\mathbf{q}, \mathbf{q})]\!]\,(t) && \text{par définition de } \mathbf{e} \\
&= [\![\mathbf{q}]\!]\,(\mathbf{q}, t) && \text{par définition d'un spécialiseur} \\
&= [\![\mathbf{p}]\!]\,\big(\mathit{spec}(\mathbf{q}, \mathbf{q}), t\big) && \text{par définition de } \mathbf{q} \\
&= [\![\mathbf{p}]\!]\,(\mathbf{e}, t) && \text{par définition de } \mathbf{e}
\end{aligned}$$

On conclut que $\mathbf{e}$ satisfait le théorème. □

On note que la démonstration précédente utilise essentiellement deux propriétés. D'une part l'existence d'un spécialiseur **spec**, et d'autre part le fait qu'il est possible de composer les exécutions du spécialiseur **spec** et du programme **p**. On remarque que la présence d'un auto-interpréteur n'est pas nécessaire.

Compilation. Comme la construction des points fixes est uniforme, le théorème de récursion de Kleene fournit un compilateur de point fixe. C'est à dire un programme **comp** tel que si on lui fournit en entrée un programme $\mathbf{p} \in \mathbb{D}$, il retourne un point fixe de la fonction de $[\![\mathbf{p}]\!]$. Ce compilateur s'exprime de la manière suivante.

$$\begin{array}{l} work = \circ\,(\mathbf{k}, work)\,; \\ work = \circ\,(work\,, work)\,; \\ \mathbf{spec} \end{array} \qquad (\mathbf{comp})$$

Pour tout programme $\mathbf{p} \in \mathbb{D}$, on a $[\![\mathbf{comp}]\!]\,(\mathbf{p}) = \mathit{spec}(\mathbf{k}\,;\mathbf{p},\mathbf{k}\,;\mathbf{p})$. On a bien le point fixe exhibé dans la démonstration du théorème 2.

Langages acceptables. Tout langage de programmation acceptable φ satisfait le théorème de récursion. Pour prouver cette propriété il suffit de prendre une bijection calculable $\mathcal{F}$ telle que pour tout programme $\mathbf{q} \in \mathbb{D}$ on a $\varphi_{\mathcal{F}(\mathbf{q})} = [\![\mathbf{p}]\!]$. On rappelle que l'existence d'une telle fonction est assurée par le théorème d'isomorphisme de Rogers. On prend $\mathbf{r} \in \mathbb{D}$ un programme de `While` calculant $\mathcal{F}$. Soit $\mathbf{p} \in \varphi$ un programme de φ, on pose $\mathbf{p}'$ le programme suivant où $x \in \mathbb{X}$ est une nouvelle variable et $\mathbf{q} = \mathcal{F}\,(\mathbf{p})$.

$$\begin{array}{l} x = tl\,(work)\,; \\ work = \mathbf{r}(hd\,(work))\,; \\ \mathbf{q} \end{array}$$

On observe que pour toute donnée $t, u \in \mathbb{D}$ ce programme satisfait $[\![\mathbf{p}']\!]\,(t,u) = \varphi_{\mathbf{p}}(\mathcal{F}\,(t), u)$. En appliquant le théorème de récursion au programme $\mathbf{p}'$ on obtient un point fixe $\mathbf{e}'$ tel que pour tout donnée $t \in \mathbb{D}$ on a $[\![\mathbf{e}']\!]\,(t) = [\![\mathbf{p}']\!]\,(\mathbf{e}', t)$. On pose $\mathbf{e} = \mathcal{F}\,(\mathbf{e}')$ et on observe

$$\begin{array}{lll} \varphi_{\mathbf{e}}(t) = [\![\mathbf{e}']\!]\,(t) & & \text{par définition de } \mathbf{e} \text{ et de } \mathcal{F} \\ \quad = [\![\mathbf{p}']\!]\,(\mathbf{e}', t) & & \text{par définition de } \mathbf{e}' \\ \quad = \varphi_{\mathbf{p}}\mathcal{F}\,(\mathbf{e}'), t & & \text{par définition de } \mathbf{p}' \\ \quad = \varphi_{\mathbf{p}}\mathbf{e}, t & & \text{par définition de } \mathcal{F} \end{array}$$

On conclut que φ satisfait le théorème de récursion.

De manière plus classique, nous aurions pu montrer cette propriété en utilisant les axiomes d'un langage de programmation acceptable. Néanmoins, le lecteur averti aura remarqué que notre approche ne fait pas directement appelle à la thèse de

Church-Turing. Le seul point non constructif réside dans l'existence du programme **r** qui est plus lié à l'existence effective du langage φ qu'à la satisfaction du théorème de récursion dans ce langage.

Temps d'exécution d'un point fixe. Par la suite, nous faisons un usage intensif du théorème de récursion en tant que compilateur de programme. Par conséquent, il est intéressant d'avoir une évaluation de l'efficacité des points fixes produits. En particulier, cette compilation sera vue comme la transformation d'une spécification, le programme en entrée, en un programme satisfaisant cette spécification, le point fixe en sortie. Ainsi, nous voulons comparer le temps d'exécution du point fixe avec celui de la spécification.

Tout d'abord nous évaluons le temps d'exécution du programme **k**. On rappelle que ce programme s'écrit

$$\begin{array}{l} x = tl\,(work)\,; \\ work = \mathbf{spec}(hd\,(work), hd\,(work))\,; \\ work = \circ\,(work\,, x)\,; \\ x = \bullet \end{array} \qquad (\mathbf{k})$$

D'après la définition de T, pour toutes données $t, u \in \mathbb{D}$ nous avons $T_{\mathbf{k}}(t,u) = T_{\mathbf{spec}}(t,t) + 18$. On rappelle que **spec** s'écrit

$$work = \circ\,(;, \circ(=, \underline{work}, \textbf{\textit{quote}}, tl\,(work)), hd\,(work)) \qquad (\mathbf{spec})$$

On obtient $T_{\mathbf{spec}}(t,t) = 16$ et on conclut que $T_{\mathbf{k}}(t) = 34$. La valeur numérique n'a pas grande importance, on retiendra que le programme **k** s'exécute en temps constant.

Prenons un programme $\mathbf{p} \in \mathbb{D}$. Le compilateur de point fixe **comp** fournit le point fixe suivant noté **e**.

$$work = \circ\,(\mathbf{k}\,;\mathbf{p}, work)\,;\mathbf{k}\,;\mathbf{p}$$

Ainsi pour toute donnée $t \in \mathbb{D}$ on a

$$T_{\mathbf{e}}(t) = 4 + T_{\mathbf{k}}(\mathbf{k}\,;\mathbf{p}, t) + T_{\mathbf{p}}(\mathbf{e}, t) = T_{\mathbf{p}}(\mathbf{e}, t) + 38$$

On observe que le temps d'exécution d'un point fixe dépend essentiellement du temps d'exécution du programme fixé. On retiendra que le temps d'exécution du programme fixé est identique au temps d'exécution du programme à une constante près.

3.4 Applications

Le théorème d'isomorphisme de Rogers [Rog58] semble être l'une de ses seules applications positives du théorème de récursion. En effet, la majeure partie des autres applications consistent en la construction de problèmes indécidables.

Cette aspect a été mis en avant par [Jon91] où Jones s'interroge sur le sens de ce théorème dans le cadre de la théorie de la programmation. Les autres piliers de la théorie de la calculabilité, que sont le théorème d'énumération et le théorème d'itération, trouvent tout deux un sens pertinent. Ce premier stipule l'existence d'auto-interpréteurs et ce deuxième assure l'existence de spécialiseurs. La combinaison de ces deux objets conduit à la notion de compilation qui est centrale en théorie de la programmation. De plus la notion de spécialisation a été largement étudié comme outil d'optimisation de programmes [JGS93].

Le théorème de récursion de Kleene ne semble pas avoir trouvé une telle place. On a souvent essayé de l'utiliser pour construire des programmes récursifs, mais cet emploi ne semble pas pertinent [HNTJ89]. Pour illustrer cela, nous donnons un exemple de programme récursif construit par application du théorème de récursion. On pose le programme $\mathbf{p}$ suivant.

$$
\begin{array}{l}
[x, y, z] = work\ ; \\
if\ (tl\ (work))\{ \\
\ \ y = \mathbf{univ}(x, y)\ ; \\
\ \ z = \mathbf{univ}(x, z)\ ; \\
\} \\
else\ \{ \\
\ \ work = \bullet\ ; \\
\}\ ; \\
work = \circ\ (z, y)
\end{array}
$$

Pour tout programme $\mathbf{q} \in \mathbb{D}$ et toute donnée $t_1, t_2 \in \mathbb{D}$ le programme $\mathbf{p}$ satisfait

$$[\![\mathbf{p}]\!]\ (\mathbf{q}, t_1, t_2) = \circ([\![\mathbf{q}]\!]\ (t_2)\ , [\![\mathbf{q}]\!]\ (t_1))$$

Le théorème de récursion fournit un programme $\mathbf{e}$ tel que

$$[\![\mathbf{e}]\!]\ (t_1, t_2) = \circ([\![\mathbf{e}]\!]\ (t_2)\ , [\![\mathbf{e}]\!]\ (t_1))$$

On observe que le programme $\mathbf{e}$ s'appelle récursivement afin de parcourrir l'arbre fourni en entrée et d'intervertir toutes ses branches.

Nous évaluons l'efficacité du programme $\mathbf{e}$. Habituellement, on considère qu'un auto-interpréteur engendre un surcoût linéaire, c'est à dire qu'il existe une constante $\alpha_{\mathbf{univ}} > 1$ telle que pour tout programme $\mathbf{p} \in \mathbb{D}$ et toute donnée $t \in \mathbb{D}$ on a

$$T_{\mathbf{univ}}(\mathbf{p}, t) \leq \alpha_{\mathbf{univ}} \cdot T_{\mathbf{p}}(t)$$

En combinant cette équation avec les temps d'exécution d'un point fixe nous obtenons dans l'exemple précédent pour tout données $t \in \mathbb{D}$ tel que $t = \circ(t_1, t_2)$

$$
\begin{aligned}
T_{\mathbf{e}}(t_1, t_2) &= O(T_{\mathbf{p}}(\mathbf{e}, t_1, t_2)) \\
&= O(\alpha_{\mathbf{univ}} \cdot T_{\mathbf{e}}(t_1) + \alpha_{\mathbf{univ}} \cdot T_{\mathbf{e}}(t_2)) \\
&= O(|t| \cdot \alpha_{\mathbf{univ}}^{d(t)})
\end{aligned}
$$

Où $d(t)$ est la profondeur de l'arbre t.

On constate un surcoût exponentiel en la profondeur de l'entrée, ce qui n'est pas acceptable en pratique. Néanmoins, il faut noter que ce surcoût est plus lié à l'utilisation d'un auto-interpréteur qu'à la construction du point fixe. D'ailleurs, en utilisant un interpréteur intégré au langage dont le temps d'exécution satisfait $T_{\mathbf{univ}}(\mathbf{p}, t) = T_{\mathbf{p}}(t) + \alpha$ pour une certaine constante $\alpha \in \mathbb{N}$, ce surcoût disparaît. On pourra consulter [Jon94, HNTJ89, Jon91] pour plus de détails sur ces aspects.

Auto-reproduction. Une autre application que l'on rencontre dans la plupart des ouvrages de référence est la construction de Quines. Du nom d'un philosophe ayant étudié le phénomène d'auto-référence, un *Quine* est un programme tel que pour toute entrée, sa sortie est égale à son propre code. Plus formellement, un programme $\mathbf{p} \in \mathbb{D}$ est un Quine si pour toute donnée $t \in \mathbb{D}$ on a $[\![\mathbf{p}]\!](t) = \mathbf{p}$.

Pour construire un Quine il suffit d'appliquer le théorème de récursion à la première projection. Soit $\mathbf{p}$ le programme $work = hd\,(work)$, on observe que pour toutes données $u, t \in \mathbb{D}$ on a $[\![\mathbf{p}]\!](u, t) = u$. Le théorème de récursion fournit un programme $\mathbf{e}$ tel que pour toute donnée $t \in \mathbb{D}$ on a $[\![\mathbf{e}]\!](t) = [\![\mathbf{p}]\!](\mathbf{e}, t) = \mathbf{e}$. Par conséquent, le point fixe $\mathbf{e}$ est un Quine.

Les Quines et les virus informatiques sont des objets très proches. Depuis cette observation plusieurs auteurs avaient pressenti le théorème de récursion comme un élément important dans la modélisation des virus informatiques. Ce lien est notamment souligné par Rogers [Rog67], Adleman [Adl88], Odifreddi [Odi89] et Smullyan [Smu94] ou plus récemment par Filiol [Fil05].

Le théorème de récursion est non seulement au cœur de la définition que nous adoptons pour modéliser les virus informatiques, mais nous utilisons aussi comme un outil constructif, une dimension qui manquait encore à ce théorème.

Virologie informatique

4 Fondements de la virologie informatique

Ce chapitre retrace la naissance de la virologie informatique. Nous commentons ce chemin théorique au travers des travaux de von Neumann [vN66], Cohen [Coh86, Coh88] et Adleman [Adl88]. Ce parcours met en avant les liens qu'entretiennent cette discipline et les modèles de calcul. De plus, il nous permet d'identifier les caractéristiques des virus informatiques. Pour compléter cet exposé nous invitons le lecteur à consulter [Fil05] où Filiol décrit cette évolution non seulement du point de vue théorique mais aussi en considérant des aspects pratiques et économiques.

Les automates cellulaires auto-reproducteurs de von Neumann constituent le point de commencement. La problématique associée à ce modèle de calcul préfigure plusieurs éléments clés de cette discipline, comme la duplication et la mutation. Cohen fut le premier à identifier la notion de virus informatique. En s'inspirant des travaux de von Neumann il pose les premières pierres de la discipline. Son approche fait encore référence aujourd'hui. Quand à Adleman, on lui doit la formalisation de la notion d'infection et la majeur partie de la terminologie utilisée par la communauté en matière de comportements viraux. Il a aussi préconisé l'emploi de la théorie de la calculabilité comme cadre formel pour la virologie informatique.

4.1 Automates cellulaires de von Neumann

Pendant les années 1940, von Neumann étudie le modèle des automates cellulaires afin de décrire le vivant par des procédés mécaniques. Son modèle se résume à un système dynamique composé de cellules interagissant entre elles afin de faire évoluer la configuration globale. Il s'interroge principalement sur la capacité d'un automate cellulaire à engendrer d'autres automates. En d'autres termes, il veut savoir si dans son formalisme un amas de cellules peut évoluer de façon à ce que l'on obtienne deux amas identiques au premier. On voit ici l'analogie avec le phénomène biologique nommé mitose.

Il répond par l'affirmative à cette question. En voyant un automate cellulaire comme un programme et son évolution comme une exécution, von Neumann préfigure la virologie informatique : il prouve l'existence de programmes capables de se répliquer à l'identique.

La problématique de von Neumann est duale à celle de la virologie informatique.

Avec les automates cellulaires von Neumann compte montrer que le vivant peut être vu comme une grande machine. Pour cela il mécanise une propriété emblématique de la biologie : l'auto-reproduction. Quand à la virologie informatique, elle est construite sur l'observation suivante : il existe des programmes auto-reproducteurs capables de se disséminer au travers des systèmes informatiques. La problématique est alors de définir un modèle mathématique permettant d'étudier ce phénomène. On comprend que cette discipline se situe en aval des travaux de von Neumann : nous ne prouvons pas l'existence d'objets auto-reproducteurs, nous partons du constat de leur existence. D'un point de vue épistémologique, il est intéressant de faire un détour par son modèle afin de reprendre les mécanismes qu'il a identifiés.

Automates cellulaires. Un automate cellulaire est constitué d'une grille de cellules et de règles de transition commandant le changement d'état de chaque cellule en fonction de son état courant et des états des cellules adjacentes. L'évolution d'un automate cellulaire correspond à l'application itérative des règles de transition.

Nous donnons une définition formelle dans le cadre d'une grille à deux dimensions. Soit $\mathbb{Q}$ un ensemble d'états, la configuration d'un automate cellulaire est une fonction $c : \mathbb{Z}^2 \to \mathbb{Q}$. Chaque couple d'entiers relatifs $(i, j) \in \mathbb{Z}^2$ représente une cellule et $c(i, j)$ donne l'état de cette cellule. Les règles de transition sont représentées par une fonction $\delta : \mathbb{Q}^5 \to \mathbb{Q}$ donnant le nouvel état d'une cellule en fonction de son état courant et des états des quatre cellules adjacentes. On a bien 5 états en entrée. On étend la fonction δ sur l'ensemble des configurations par la notation suivante.

$$\delta(c)(i,j) = \delta(q_1, q_2, q_3, q_4, q_5) \quad \text{avec} \quad \begin{array}{llllll} q_1 & = & c(i,j) & & & \\ q_2 & = & c(i-1,j) & q_3 & = & c(i+1,j) \\ q_4 & = & c(i,j-1) & q_5 & = & c(i,j+1) \end{array}$$

On dit que $\delta(c)$ est la configuration obtenue après un pas de l'automate cellulaire depuis la configuration c. Pour tout entier $n \in \mathbb{N}$ on note $\delta^n(c)$ la configuration obtenue après n pas successifs.

On se munit d'un état particulier $q_0 \in \mathbb{Q}$ qualifié de quiescent. Une cellule est quiescente si elle se trouve dans l'état q_0. On pose deux restrictions. D'une part, toute cellule entourée de cellules quiescentes devient quiescente : pour tout état $q \in \mathbb{Q}$ on a

$$\delta(q, q_0, q_0, q_0, q_0) = q_0$$

D'autre part on se limite aux configurations ne comportant qu'un nombre fini de cellules non quiescentes : les configurations finies. On observe que la première restriction implique que l'évolution d'une configuration finie conduit forcément à une configuration finie.

Automates cellulaires auto-reproducteurs. Une configuration $c : \mathbb{Z}^2 \to \mathbb{Q}$ est auto-reproductrice relativement à la fonction de transition δ si il existe un entier $n \in \mathbb{N}$ et un couple d'entiers relatifs $(k, l) \in \mathbb{Z}$ tels que pour tout couple d'entiers relatifs $(i, j) \in \mathbb{Z}$ on a les propriétés suivantes.

– La configuration initiale est reproduite avec un décalage (k, l) sur la grille :

$$c(i,j) \neq q_0 \Rightarrow \big(\delta^n(c)(i,j) = c(i,j) \text{ et } \delta^n(c)(k+i, l+j) = c(i,j)\big)$$

– La zone de reproduction était précédemment quiescente :

$$c(i,j) \neq q_0 \Rightarrow c(k+i, l+j) = q_0$$

– Les cellules hors de la zone de reproduction sont quiescentes :

$$\big(c(i,j) = q_0 \text{ et } c(k+i, l+j) = q_0\big) \Rightarrow \delta^n(c)(k+i, l+j) = q_0$$

Von Neumann [vN66] prouve l'existence d'une configuration et d'une fonction de transition satisfaisant ces contraintes. Sa construction est fondée sur deux éléments : un constructeur universel et une digonalisation. Un constructeur universel est un automate capable de reproduire tout automate depuis une description de ce dernier. On remarque que cet objet est intuitivement très proche d'un auto-interpréteur. La diagonalisation consiste à prendre une configuration associant un automate universel et sa description. Ainsi l'automate universel se construit lui-même en utilisant sa propre description.

En observant sa construction, von Neumann pense que la présence d'un automate universel est indispensable à l'auto-reproduction. Cette hypothèse est contredite par les constructions de Codd [Cod68], de Langton [Lan84] et bien d'autres par la suite. Ce résultat corrobore la remarque faite lors de la démonstration du théorème de récursion : la présence d'un auto-interpréteur ne semble pas nécessaire aux constructions auto-référentes. Cette idée sera développée au chapitre 7.

Von Neumann s'est aussi interrogé sur les possibilités de transformation d'un automate cellulaire au cours de sa reproduction. Il compte ainsi mécaniser le phénomène biologique d'évolution du matériel génétique. En énonçant ce principe dans un cadre formel, il préfigure la mutation des codes viraux. Cette notion est cruciale en virologie informatique, à l'heure actuelle la majeure partie des attaques virales utilise des techniques de mutation afin de rendre leur détection plus ardue.

4.2 Le formalisme de Cohen

Nous revenons à la virologie informatique. Cette discipline a vu le jour avec la thèse d'état de Cohen [Coh86] où l'on trouve la première description formelle des virus informatiques. D'ailleurs, on doit cette terminologie à Adleman, le directeur de thèse de Cohen. A cette époque, en 1986, seuls quelques cas de virus ont été observés. On comprend que Cohen est un pionnier dans le domaine.

Pour construire son formalisme, Cohen propose une définition intuitive qui fait référence dans la communauté.

> Un virus informatique est un programme qui peut infecter d'autres programmes en les modifiant dans le but d'insérer une copie possiblement évoluée de lui-même.

On retiendra trois éléments importants. En premier lieu, un virus est un programme. Deuxièmement, un virus est capable d'infecter un programme hôte. Enfin, la forme d'un virus peut évoluer au cours de sa reproduction.

Machines de Turing. Cohen donne un modèle mathématique de sa définition dans le cadre des machines de Turing. On rappelle qu'une telle machine est composée d'un ruban comportant une infinité de cases et d'une tête de lecture. Chaque case peut accueillir une lettre. Avec Σ un alphabet et $\mathbb{Q}$ un ensemble d'états, la configuration d'une machine de Turing est représentée par un triplet $c = (\triangle, q, \mathit{tape})$ où

- $\triangle \in \mathbb{N}$ est la position de la tête sur le ruban.
- $q \in \mathbb{Q}$ est l'état de la machine.
- $\mathit{tape} : \mathbb{N} \to \Sigma$ représente le ruban : $\mathit{tape}(n)$ désigne la lettre présente sur la n-ème case du ruban.

Par la suite $\mathbb{C}$ désigne l'ensemble des configurations.

Pour tous entiers $i, j \in \mathbb{N}$ tels que $i \leq j$ on note $\mathit{tape}(i \cdots j)$ le mot obtenu par la concaténation des lettres $\mathit{tape}(i), \cdots, \mathit{tape}(j)$. On dit qu'un mot $\nu \in \Sigma^*$ est *en tête* d'une configuration $(\triangle, q, \mathit{tape})$ si $\mathit{tape}(\triangle), \cdots, \mathit{tape}(\triangle + |\nu|) = \nu$. On se munit d'un état particulier note q_0 représentant l'*état initial* des machines de Turing. Une configuration est dite initiale si son état est q_0.

La machinerie est décrite par un ensemble de règles de transition. En fonction de la configuration courante de la machine, ces règles commandent

- l'écriture d'une nouvelle lettre sur la case indiquée par la tête,
- le déplacement de la tête sur l'une des deux cases adjacentes,
- la transition de la machine vers un nouvel état.

On représente une machine de Turing par un triplet $M = (\mathit{mov}, \mathit{trans}, \mathit{write})$ où

- $\mathit{write} : \mathbb{C} \to \Sigma$ donne le lettre à écrire sur la case,
- $\mathit{mov} : \mathbb{C} \to \{-1, 0, 1\}$ donne le déplacement de la tête,
- $\mathit{trans} : \mathbb{C} \to \mathbb{Q}$ donne le nouvel état.

Pour toute configuration $c = (\triangle, q, \mathit{tape})$, on note $M(c)$ la configuration obtenue après un pas d'exécution de la machine M depuis la configuration c.

$$M(c) = (\triangle + \mathit{mov}(c), \mathit{trans}(c), \mathit{tape}[\triangle \mapsto \mathit{write}(c)])$$

Pour tout entier n on note $M^n(c) = (\triangle^n, q^n, \mathit{tape}^n)$ la configuration obtenue après n pas successifs.

Les virus de Cohen. Dans ses travaux [Coh86, Coh88], Cohen modélise un virus informatique par un ensemble de codes viraux se propageant sur le ruban d'une machine de Turing. Un ensemble de mots $\mathbb{V}_c \subset \Sigma^*$ est un *ensemble viral* si pour tout *code viral* $\nu \in \mathbb{V}_c$ et toute configuration initiale $c = (\triangle, q_0, \textit{tape})$ ayant ν en tête, il existe un deuxième code viral $\omega \in \mathbb{V}_c$, deux temps d'exécution $n, m \in \mathbb{N}$, et une position sur le ruban i tels que

- $\textit{tape}^n(i \cdots i + |\omega|) = \omega$: après n pas, un nouveau code viral de l'ensemble viral se trouve sur le ruban à la position i.
- $i \notin \{\triangle, \cdots, \triangle + |\nu|\}$: ce code viral se trouve sur un emplacement distinct de celui où se trouvait ν.
- $m \leq n$ et $\triangle^m \in \{i, \cdots, i + |\omega|\}$: au cours des n pas d'exécution, la tête de la machine a visité l'emplacement où se trouve ω.

De manière intuitive, tout code viral en tête d'une configuration initiale mène à la construction d'un autre code viral.

Les avancées. Si on fait un parallèle entre les cases d'une machine de Turing et les cellules d'un automate, on observe que la définition d'un ensemble viral est très proche de la notion d'automate cellulaire auto-reproducteur. La première propriété est analogue à la duplication d'un amas de cellules mais avec une possibilité de mutation. De plus, on retrouve la contrainte interdisant le chevauchement de l'entité originale et de la zone de reproduction. Par contre Cohen s'abstrait des contraintes de quiescence du voisinage, cela afin d'intégrer un mécanisme d'infection : le voisinage représente le programme hôte.

La différence majeure avec les recherches de von Neumann est le but affiché par Cohen. La notion d'ensemble viral met intégre la notion de mutation des virus ce qui met en avant la difficulté liée à la détection. En particulier, Cohen montre que la détection des virus est un problème indécidable [Coh88]. Par cette démarche, il place les infections virales comme une menace réelle pour les systèmes informatiques. Pour étayer ses résultats théoriques, il mène des expériences sur des systèmes réels. Les résultats sont catastrophiques dans le sens où ils mettent à jour la vulnérabilité des infrastructures de l'époque. L'impact est tel que la poursuite de ces expériences est interdite.

Par ailleurs, Cohen explique que l'étude des virus informatiques ne se limite pas aux programmes malveillants. Il illustre plusieurs de ses article par des virus « bienveillants » comme celui permettant d'économiser de l'espace de stockage. Ce point incite à se focaliser sur le phénomène d'auto-reproduction et à délaisser celui trop subjectif de la malveillance.

De manière plus anecdotique, Cohen s'intéresse aux systèmes libres de tout virus. Il montre l'existence d'une machine qui n'admet aucun ensemble viral : la machine qui ne fait rien. Cette observation et ses expériences le conduisent à énoncer l'adage selon lequel, seul un système isolé est protégé des infections informatiques.

On pourra noter une seule faiblesse dans le travail de Cohen : alors qu'il entrevoit le phénomène d'infection, cette notion n'est pas explicite dans son formalisme. En effet, il faut attendre un article complémentaire [Adl88] écrit par Adleman pour avoir une formalisation plus rigoureuse des mécanismes d'infection.

4.3 Le formalisme d'Adleman

Dans [Adl88], Adleman considère la définition informelle suivante. Un virus est une fonction qui transforme tout programme en une forme infectée. Suivant son entrée, une forme infectée suit l'un des comportements suivants.

Nuisance. La forme infectée opère une tâche différente de la tâche originale.

Infection. La forme infectée accomplie la tâche originale puis infecte d'autres programmes.

Furtivité. La forme infectée accomplie la tâche originale.

Afin d'exposer son formalisme, Adleman se place dans la théorie de la calculabilité. Par conséquent, nous pouvons reprendre les notations des chapitres précédents pour exposer ses travaux.

Les virus d'Adleman. Un *scénario* permet de décrire l'évolution d'un système informatique. Les programmes sont vus comme des tâches à effectuer sur le système, l'entrée du programme représente l'état du système avant l'exécution et la sortie rend compte des modifications apportées à cet état.

Dans le *scénario d'Adleman* l'état du système est décrit par un couple $\langle \mathbf{q}, t\rangle$ où $\mathbf{q}$ est la liste des programmes présents sur le système et t est l'ensemble des autres paramètres. La liste $\mathbf{q}$ est de la forme $\langle \mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet\rangle$ où $\mathbf{q}_1, \cdots, \mathbf{q}_n$ sont des programmes pouvant être exécutés. Lorsque que l'on exécute un programme, il prend comme entrée l'état courant du système et retourne en sortie le nouvel état.

Pour toute fonction totale $\mathcal{F} : \mathbb{D} \to \mathbb{D}$ et toutes entrées $\langle \mathbf{q}, t\rangle$ et $\langle \mathbf{q}', t'\rangle$ on note $\langle \mathbf{q}, t\rangle \overset{\mathcal{F}}{\cong} \langle \mathbf{q}, t\rangle$ si

$$t = t' \qquad \mathbf{q} = \langle \mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet\rangle \qquad \mathbf{q}' = \langle \mathbf{q}'_1, \cdots, \mathbf{q}'_n, \bullet\rangle$$

où pour tout $i \in \{1, \cdots, n\}$ on a $\mathbf{q}'_i \in \{\mathbf{q}_i, \mathcal{F}(\mathbf{q}_i)\}$. De plus, on note $\langle \mathbf{q}, t\rangle \overset{\mathcal{F}}{\sim} \langle \mathbf{q}', t'\rangle$ si $\langle \mathbf{q}, t\rangle \overset{\mathcal{F}}{\cong} \langle \mathbf{q}', t'\rangle$ et $\langle \mathbf{q}, t\rangle \neq \langle \mathbf{q}', t'\rangle$.

Une fonction totale $\mathcal{A} : \mathbb{D} \to \mathbb{D}$ est un *virus d'Adleman* si pour toute entrée $\langle \mathbf{q}, t\rangle \in \mathbb{D}$ elle satisfait l'une des trois propriétés suivantes.

$$\begin{array}{ll} \text{Nuisance.} & \forall \mathbf{p}, \mathbf{r} \in \mathbb{D} : [\![\mathcal{A}(\mathbf{p})]\!](\mathbf{q}, t) = [\![\mathcal{A}(\mathbf{r})]\!](\mathbf{q}, t) \\ \text{Infection.} & \forall \mathbf{p} \in \mathbb{D} : [\![\mathcal{A}(\mathbf{p})]\!](\mathbf{q}, t) \overset{\mathcal{A}}{\sim} [\![\mathbf{p}]\!](\mathbf{q}, t) \\ \text{Furtivité.} & \forall \mathbf{p} \in \mathbb{D} : [\![\mathcal{A}(\mathbf{p})]\!](\mathbf{q}, t) = [\![\mathbf{p}]\!](\mathbf{q}, t) \end{array}$$

Pour tout programme $\mathbf{p} \in \mathbb{D}$ on dit que $\mathcal{A}(\mathbf{p})$ est la *forme infectée* de l'hôte $\mathbf{p}$ par le virus $\mathcal{A}$. On dira qu'un virus d'Adleman $\mathcal{A}$ est *effectif* si la fonction $\mathcal{A}$ est calculable.

La première branche de la définition caractérise une nuisance dans le sens où toutes les formes infectées ont le même comportement, on parle habituellement de *charge virale*. Dans la seconde branche, la forme infectée retourne une sortie identique à celle obtenue par l'exécution du programme hôte, à ceci près que certains programmes ont été remplacés par leurs formes infectées. Ici l'infection est propagée à travers le système. Dans la dernière branche, l'infection n'opère aucun changement aux fonctionnalités de l'hôte, il y a furtivité puisque aucune différence de comportement ne peut être observée.

Une classification des infections virales. La définition d'Adleman permet d'introduire une *classification des infections virales* grâce aux critères suivants. Une forme infectée $\mathcal{A}(\mathbf{p})$ est

Pathogène. S'il existe une entrée $\langle \mathbf{q}, t\rangle \in \mathbb{D}$ telle que pour tout programme $\mathbf{r} \in \mathbb{D}$ on a $[\![\mathcal{A}(\mathbf{p})]\!](\mathbf{q}, t) = [\![\mathcal{A}(\mathbf{r})]\!](\mathbf{q}, t)$. C'est un cas de nuisance.

Contagieuse. S'il existe une entrée $\langle \mathbf{q}, t\rangle \in \mathbb{D}$ telle que $[\![\mathcal{A}(\mathbf{p})]\!](\mathbf{q}, t) \overset{\mathcal{A}}{\sim} [\![\mathbf{r}]\!](\mathbf{q}, t)$. C'est un cas d'infection.

Bénine. Si elle n'est ni pathogène ni contagieuse. En d'autres termes $\mathcal{A}$ n'a pas changé la sémantique de son hôte, on a $[\![\mathcal{A}(\mathbf{p})]\!] = [\![\mathbf{p}]\!]$.

Troyenne. Si elle est pathogène sans être contagieuse.

Porteur sain. Si elle est contagieuse sans être pathogène.

Virulente. Si elle est contagieuse et pathogène.

Ensuite, on peut qualifier le comportement d'un virus suivant le comportement des formes infectées qu'il engendre. Un virus est

Bénin. Si toutes les formes infectées sont bénines.

Epien. S'il existe une forme infectée pathogène et si aucune n'est contagieuse.

Disséminateur. S'il existe une forme infectée contagieuse et si aucune n'est pathogène.

Malicieux. S'il existe une forme infectée contagieuse et une forme infectée pathogène.

Les avancées. On observe une rupture avec les approches de von Neumann et Cohen. Au lieu d'identifier un mécanisme d'auto-reproduction dans un modèle donné, Adleman montre que ce phénomène est inhérent à tout modèle de calcul raisonnable. Pour arriver à cette conclusion, il s'abstrait de toute machinerie et travaille sur les fonctions partielles récursives. Ensuite, il lui a suffi d'employer le théorème de récursion pour exhiber des exemples.

Par ce formalisme, Adleman souligne que tout système informatique raisonnable est exposé aux attaques virales. En effet, nous connaissons aujourd'hui des cas d'infection sur la majorité des systèmes qu'il soit Apple, Microsoft, Unix ou même embarqué.

Par ailleurs, alors que Cohen avait éludé la notion d'infection, Adleman en fait la pierre angulaire de son formalisme. En rendant cette notion explicite, il établit une taxinomie des comportements viraux. En particulier, il montre qu'un virus ne

se réduit pas à un programme auto-reproducteur, il peut aussi inclure une charge virale.

Par la mise en avant de la possibilité de nuisance des infections informatiques, Adleman renouvelle l'avertissement de Cohen. Dans [Adl88], il réduit le problème de la détection des virus à celui de la prouvabilité d'une formule, cela lui permet de reprendre les résultats classiques d'indécidabilité et d'inséparabilité récursive. Il insiste ainsi sur la difficulté liée à la détection des infections informatiques. Malheureusement, ces travaux n'ont pas les échos attendus et la recherche théorique dans cette discipline est très pauvre par la suite.

Les limitations. Le formalisme d'Adleman possède plusieurs limitations. En premier lieu la notion de mutation y est difficile à modéliser. Il proposait de considérer un ensemble de fonctions se propageant les unes les autres. Cette idée fut mise en pratique par Zuo et Zhou dans l'article [ZZ04] par l'ajout d'un argument aux virus d'Adleman. Ainsi un virus de polymorphe peut être vu comme une énumération de virus d'Adleman $\{\mathcal{A}_i\}_{i\in\mathbb{D}}$. Toutefois, même si cette amélioration permet de modéliser un virus capable de muter sa procédure d'infection, elle ne permet pas de rendre compte de la mutation du code viral en lui-même. En effet il existe des codes viraux mutants dont la procédure d'infection reste inchangée. Nous verrons cela au chapitre 6.

Par ailleurs, Adleman regrette lui-même de ne pas être capable de décrire les programmes se répliquant à l'identique sans mécanisme d'infection. Comme ses virus sont purement fonctionnels, ils doivent nécessairement infecter un hôte pour se reproduire. Cette contrainte empêche notamment la description des vers informatiques.

Une dernière limitation est liée au scénario d'infection. Selon Adleman une forme infectée exécute son programme hôte puis infecte d'autre programmes, on parle de *post-infection*. En pratique, on observe souvent l'inverse : une forme infectée propage l'infection puis exécute son programme hôte, on parle de *pré-infection*. Cette limitation a été soulignée par Zuo et Zhou [ZZ04], néanmoins ils n'y apportent pas de réelle solution. Ils montrent seulement que ce deuxième scénario peut être modélisé de manière analogue. Confronté à un autre scénario de propagation, nous sommes démunis.

4.4 Conclusions

Nous reprenons les points essentiels de ce chapitre. Les travaux de von Neumann nous montrent qu'un programme auto-reproducteur peut être construit par un mécanisme de digonalisation. Pour cela, il suffit de prendre un programme capable de se dupliquer en utilisant sa propre description. De plus, en évoquant les boucles de Codd et Langton, nous avons à nouveau remarqué qu'un auto-interpréteur n'est

pas nécessaire à l'auto-reproduction.

En étudiant le formalisme de Cohen nous avons identifié trois caractéristiques essentielles à la modélisation des virus informatiques.

- Un virus est un programme.
- Un virus peut infecter d'autres programmes.
- La forme d'un virus peut évoluer au cours de sa reproduction.

De plus, nous avons noté que même si les virus sont souvent considérés comme des programmes malveillants, il en existe des bienveillants. Si l'on veut réaliser une étude rigoureuse, il faut dans un premier temps laisser cette notion de coté.

Adleman ouvre la voie de la virologie abstraite en étudiant les virus informatiques dans le cadre de la calculabilité. Cette approche est importante car on constate que la notion d'infection virale est indépendante du modèle de calcul considéré. Par ailleurs, sa modélisation établit que la notion d'infection est un élément clé. Néanmoins, il ne faut pas négliger le fait qu'un virus est un programme. Cette caractéristique semble indispensable pour la description de phénomène de mutation. L'association de ces deux remarques fait apparaître une dualité entre code viraux et mécanismes d'infection. Enfin, le scénario considéré pour décrire les infections virales doit être suffisamment générique pour ne pas limiter la portée de notre étude.

5 Virologie informatique abstraite

Au travers des définitions présentées au chapitre 4 nous avons observé deux visions des virus informatiques. D'un coté, Cohen met en avant la nature de programme auto-reproducteur. De l'autre, Adleman se focalise sur la notion d'infection et prend comme modèle une propriété fonctionnelle. Nous pensons que ces deux points de vue sont non seulement complémentaires mais aussi nécessaires à une définition robuste. L'objectif de ce chapitre est de regrouper les différentes caractéristiques que nous avons identifiées autour d'un formalisme souple, générique et constructif.

Notre développement se déroulera en deux temps. Premièrement nous adoptons une démarche inductive : par la construction d'un virus parasite nous mettons en évidence la dualité qu'entretiennent les deux visions précitées. Depuis cette observation, nous établissons une définition fondée non seulement sur cette dualité mais aussi sur le théorème de récursion. Comme ce théorème permet de construire des programmes auto-référents, en le plaçant au cœur de notre définition nous lui donnons le rôle de compilateur de virus. Cette approche à l'avantage de produire un formalisme constructif.

Deuxièmement nous justifions cette démarche inductive en montrant la généralité de notre approche. La cohabitation des notions de code viral et d'infection permet d'identifier deux mécanismes de propagation d'un virus. Le premier que nous nommons *blueprint* rend compte d'un virus comme une empreinte se propageant à l'intérieur d'un système. Ici seul le code viral se propage et le phénomène d'infection reste implicite. Dans le second mécanisme, nommé *Smith*, la totalité du système viral est propagé afin de rendre explicite la propriété d'infection.

Chacune de ces méthodes de reproduction peut être associée à une forme particulière du théorème de récursion. Même si nous nous accordons avec Thompson [Tho84] et Ludwig [Lud98], pour dire que la construction de programmes auto-reproducteur n'est pas une chose triviale, nous montrons qu'elle peut être systématisée par l'utilisation de ces théorèmes de récursion. Cette observation nous amènera à la notion de compilateurs de virus.

Enfin nous montrons que notre définition est suffisamment souple pour inclure celle d'Adleman de manière canonique. Nous allons aussi au delà en exhibant une construction virale ne rentrant pas dans les critères d'Adleman.

Au long de ce chapitre, nous gardons une approche pédagogique en présentant des programmes du langage `While` illustrant nos constructions théoriques. L'objectif est de démystifier la création des virus informatiques. Nous pensons que cette dissection minutieuse des constructions virales est nécessaire à la compréhension

des propriétés sous-jacentes. De notre point de vue, ce travail est un passage obligé pour l'élaboration de solutions antivirales fondées sur des bases solides.

5.1 Une définition des virus informatiques

Le scénario. Nous nous plaçons dans un scénario analogue à celui d'Adleman à ceci près que nous ne distinguons pas les programmes des données. Le langage `While` modélise un système informatique. Les programmes représentent des tâches exécutées sur le système. L'entrée d'un programme décrit l'état du système avant l'exécution du programme et la sortie représente l'état après cette exécution.

Prenons un exemple où le système contient une liste de programmes $\mathbf{q}_1, \cdots, \mathbf{q}_n \in \mathbb{D}$ représentée par l'entrée $\circ(\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet)$. Dans ce contexte le programme suivant opère la copie du premier programme de la liste.

$$work = \circ\,(hd\,(work), work) \qquad (\mathbf{copy})$$

En effet, on a $[\![\mathbf{copy}]\!]\,(\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet) = \circ(\mathbf{q}_1, \mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet)$.

Le virus parasite. Comme son nom l'indique un virus parasite s'insère à l'intérieur d'un programme hôte et le modifie afin de propager l'infection. Si on exécute un programme infecté par un virus parasite alors le virus infecte un nouvel hôte puis exécute le programme hôte. Ce virus a pour unique but de se propager tout en préservant le système infecté dans le sens où les fonctionnalités des hôtes sont conservées. Néanmoins, il serait possible d'adjoindre une charge virale déclenchée suivant certaines conditions sur l'entrée.

D'un point de vu plus formel, l'équation suivante décrit le comportement d'un virus parasite. Pour tous programmes $\mathbf{p}, \mathbf{q}_1, \cdots, \mathbf{q}_n, \in \mathbb{D}$

$$[\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!]\,(\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet) = [\![\mathbf{p}]\!]\,(\mathcal{B}(\mathbf{v}, \mathbf{q}_n), \cdots, \mathcal{B}(\mathbf{v}, \mathbf{q}_1), \bullet) \qquad (5.1)$$

Le programme $\mathbf{v} \in \mathbb{D}$ est nommé *code viral*. C'est une empreinte qui est propagée au travers des hôtes. La fonction $\mathcal{B} : \mathbb{D} \to \mathbb{D}$ est nommée fonction de propagation, elle décrit comment l'empreinte $\mathbf{v}$ est insérée à l'intérieur d'un hôte. Le programme $\mathcal{B}(\mathbf{v}, \mathbf{p})$ est nommée la *forme infectée* du programme $\mathbf{p}$ par le code viral $\mathbf{v}$. Dans l'équation précédente, toute forme infectée $\mathcal{B}(\mathbf{v}, \mathbf{p})$ exécute l'hôte original $\mathbf{p}$ sur l'entrée résultant de l'infection des programmes du système par le code $\mathbf{v}$ et la fonction de propagation $\mathcal{B}$.

On remarque que l'ordre des programmes est inversé. Nous avons fait ce choix par convenance afin de pouvoir écrire un programme concis.

Il est assez simple de construire un tel virus en utilisant le théorème de récursion. Pour cela, il suffit de poser $\mathcal{B} = \mathit{spec}$ puis d'appliquer le théorème de récursion au

programme suivant.

$$\begin{array}{l} [x_{\mathbf{v}}, x_{\mathbf{p}}, x_{\mathbf{q}}] = work\,; \\ foreach\,(y\ in\ x_{\mathbf{q}})\{ \\ \quad z = \mathbf{spec}(\mathbf{v}, y)\,; \\ \quad \alpha = \circ\,(z, \alpha) \\ \}\,; \\ work = \mathbf{univ}(x_{\mathbf{p}}, \alpha) \end{array} \qquad (\mathbf{r})$$

On obtient un programme $\mathbf{v} \in \mathbb{D}$ satisfaisant pour tous programmes $\mathbf{p}, \mathbf{q}_1, \cdots, \mathbf{q}_n, \in \mathbb{D}$

$$[\![\mathbf{v}]\!]\,(\mathbf{p}, \mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet) = [\![\mathbf{p}]\!]\,\big(\mathit{spec}(\mathbf{v}, \mathbf{q}_n), \cdots, \mathit{spec}(\mathbf{v}, \mathbf{q}_1), \bullet\big)$$

Par définition du spécialiseur *spec*, on a de plus

$$[\![\mathit{spec}(\mathbf{v}, \mathbf{p})]\!]\,(\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet) = [\![\mathbf{v}]\!]\,(\mathbf{p}, \mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet)$$

En posant $\mathcal{B} = \mathit{spec}$ et en combinant les deux équations précédentes, on obtient l'équation (5.1). En d'autres termes, le code viral $\mathbf{v}$ et la fonction de propagation *spec* constituent un virus parasite.

Le mécanisme viral. L'exemple précédent fait apparaître une dualité entre le programme se propageant à travers un système et la propriétés fonctionnelles associant les programmes à leur forme infectée. Ces deux notions sont respectivement matérialisées par le code viral $\mathbf{v}$ et la fonction de propagation $\mathcal{B}$. Le code viral exprime le comportement de l'infection, il décrit comment la fonction de propagation est employée. L'association d'un code viral et d'une fonction de propagation définie une infection virale transformant les programmes en formes infectées suivant l'application $\mathbf{p} \mapsto \mathcal{B}(\mathbf{v}, \mathbf{p})$. Enfin, l'exécution d'une forme infectée est équivalente à l'exécution du code viral, cette propriété fonctionnelle garantie la propagation de l'infection.

Ce mécanisme nous permet de considérer une double interprétation de la notion de virus informatique. On y reconnaît les points de vue de Cohen et d'Adleman. D'une part un virus est un programme se propageant au travers du système informatique. D'autre part la fonction de propagation modélise le vecteur d'infection par lequel le code viral se dissémine. De manière plus formelle, nous donnons la définition suivante.

Définition 3 (Virus)**.** Soit $\mathcal{B}$ une fonction de $\mathbb{D}$ vers $\mathbb{D}$. Le programme $\mathbf{v} \in \mathbb{D}$ est un *virus* relativement à la *fonction de propagation* $\mathcal{B}$ si pour tout programme $\mathbf{p} \in \mathbb{D}$ et toute donnée $t \in \mathbb{D}$ on a

$$[\![\mathbf{v}]\!]\,(\mathbf{p}, t) = [\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!]\,(t)$$

Infection et spécialisation. On observe qu'une fonction de spécialisation est un cas particulier de fonction de propagation. Cela n'est pas anodin, en effet un code viral peut être vu comme un programme utilisant son hôte pour réaliser une certaine action. Par exemple, dans le cas d'un virus parasite, le code viral exécute le programme original de son hôte puis infecte d'autre programmes. Ainsi on peut voir l'infection d'un hôte $\mathbf{p}$ comme la spécialisation du code viral $\mathbf{v}$ pour l'hôte $\mathbf{p}$, le résultat étant la forme infectée $\mathcal{B}(\mathbf{v}, \mathbf{p})$. On remarque que tout programme est un virus relativement à toute fonction de spécialisation.

Une définition constructive. Notre définition est fondée sur le théorème de récursion, ce théorème prend ainsi le rôle d'un constructeur de virus. En effet, étant donnée une fonction de propagation semi-calculable, la fonction $\mathcal{F} : \mathbb{D} \to \mathbb{D}$ définie par $\mathcal{F}(u, \mathbf{p}, t) = \mathit{univ}(\mathcal{B}(u, \mathbf{p}), t)$ est semi-calculable comme composée de fonctions semi-calculables. Le théorème de récursion nous donne l'existence d'un programme $\mathbf{v}$ tel que

$$[\![\mathbf{v}]\!](\mathbf{p}, t) = \mathcal{F}(\mathbf{v}, \mathbf{p}, t) = [\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!](t)$$

On observe que $\mathbf{v}$ est un virus relativement à la fonction de propagation $\mathcal{B}$.

5.2 Les virus blueprint

Infection implicite. Un virus *blueprint* est un virus se recopiant sans utiliser de fonction de propagation. Ici le processus d'infection reste implicite. D'un point de vue formel, un virus blueprint est défini par son seul code viral.

Définition 4 (Virus blueprint)**.** Soit $\mathcal{F}$ une fonction de $\mathbb{D}$ vers $\mathbb{D}$. Un programme $\mathbf{v} \in \mathbb{D}$ est un virus *blueprint* relativement au *comportement* $\mathcal{F}$ si pour toute donnée $t \in \mathbb{D}$ on a

$$[\![\mathbf{v}]\!](t) = \mathcal{F}(\mathbf{v}, t)$$

Dans l'équation précédente, la fonction $\mathcal{F}$ spécifie l'interaction entre le code viral $\mathbf{v}$ et l'état du système t. C'est pour cette raison que l'on dit que $\mathcal{F}$ représente le comportement de l'infection.

En reprenant la définition 3, La construction d'un virus blueprint revient à résoudre le système d'équations suivant.

$$\begin{cases} [\![\mathbf{v}]\!](t) = [\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!](t) \\ [\![\mathbf{v}]\!](t) = \mathcal{F}(\mathbf{v}, t) \end{cases}$$

Comme un virus blueprint n'utilise pas explicitement de fonction de propagation, on peut fixer $\mathcal{B} = \mathit{spec}$ afin que la première équation soit validée. On peut alors construire un programme $\mathbf{v} \in \mathbb{D}$ satisfaisant la seconde équation par application directe du théorème de récursion.

Compilation de virus blueprint. Pour tout programme $\mathbf{r} \in \mathbb{D}$, il est possible de construire un virus blueprint relativement au comportement $[\![\mathbf{r}]\!]$. Il suffit d'appliquer le théorème de récursion pour obtenir un programme $\mathbf{v}$ tel que pour toute donnée $t \in \mathbb{D}$ on a $[\![\mathbf{v}]\!](t) = [\![\mathbf{r}]\!](\mathbf{v}, t)$.

Maintenant en voyant le programme $\mathbf{r}$ comme la spécification du comportement d'un virus blueprint, le compilateur de point fixe exposé au chapitre 3 nous fournit un compilateur de virus blueprint. En effet, le programme $\mathbf{v} = [\![\mathbf{comp}]\!](\mathbf{r})$ est un virus blueprint satisfaisant le comportement $[\![\mathbf{r}]\!]$.

Propriété 5. *Il existe un compilateur de virus blueprint.*

Démonstration. Prendre le compilateur de point fixe page 27. □

Un virus écraseur. Pour illustrer la notion de virus blueprint nous considérons un exemple concret : un virus *écraseur*. Un tel virus remplace les programmes d'un système par des copies de lui-même. A terme cette infection conduit à ce que le système ne contienne plus que des copies du virus. Des versions réelles de virus écraseurs sont décrites dans les ouvrages [Lud98, Fil05].

On peut rendre compte d'un virus écraseur par un code viral $\mathbf{v} \in \mathbb{D}$ satisfaisant pour tous programmes $\mathbf{q}_1, \cdots, \mathbf{q}_n \in \mathbb{D}$

$$[\![\mathbf{v}]\!](\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet) = \circ(\underbrace{\mathbf{v}, \cdots, \mathbf{v}}_{n \text{ fois}}, \bullet)$$

On observe que l'exécution de $\mathbf{v}$ sur une liste de programmes $\mathbf{q}_1, \cdots, \mathbf{q}_n$ conduit au remplacement de ces programmes par des copies de $\mathbf{v}$.

Cette équation rentre dans le cadre des virus blueprint. Il est très facile de construire un tel virus grâce au théorème de récursion. On se donne le programme $\mathbf{r}$ défini par

```
[x, y] = work ;          // décompose l'entrée
work = • ;               // réinitialise l'état
foreach (z ∈ y){                                   (r)
 work = ∘ (x, work) ;    // remplace chaque programme par la valeur de x
} ;
```

On observe que pour tous $\mathbf{q}_1, \cdots, \mathbf{q}_n, u \in \mathbb{D}$ le programme $\mathbf{r}$ satisfait

$$[\![\mathbf{r}]\!](t, \circ(\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet)) = \circ(\underbrace{t, \cdots, t}_{n \text{ fois}}, \bullet)$$

Le théorème de récursion donne le programme $\mathbf{v} = [\![\mathbf{comp}]\!](\mathbf{r})$ satisfaisant

$$[\![\mathbf{v}]\!](\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet) = [\![\mathbf{r}]\!](\mathbf{v}, \circ(\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet)) = \circ(\underbrace{\mathbf{v}, \cdots, \mathbf{v}}_{n \text{ fois}}, \bullet)$$

Cette exemple nous permet de comprendre en quoi le théorème de récursion peut être vu comme un compilateur de virus. Le programme **r** constitue une spécification du virus écraseur : ce programme définit comment le code viral interagit avec l'état du système. Le théorème de récursion permet de construire un virus blueprint satisfaisant cette spécification.

5.3 Les virus Smith

Reproduction par vecteur d'infection. Nous passons à une deuxième méthode virale nommée *reproduction par vecteur d'infection* [BKM06]. Les virus associés à ce mécanisme sont appelés *virus Smith*. Contrairement aux virus blueprint, les virus Smith infectent leurs hôtes par l'intermédiaire d'une fonction de propagation. De plus, cette fonction est propagée aux cotés du code viral afin d'assurer la dissémination.

Définition 6 (Virus Smith)**.** Soit $\mathcal{F}$ une fonction de $\mathbb{D}$ vers $\mathbb{D}$. Un programme $\mathbf{v}$ associé à une fonction $\mathcal{B}$ est un *virus Smith* relativement au comportement $\mathcal{F}$ si

- $\mathbf{v}$ est un virus relativement à la fonction de propagation $\mathcal{B}$
- il existe un programme $\mathbf{b} \in \mathbb{D}$ calculant $\mathcal{B}$ tel que pour tout programme $\mathbf{p} \in \mathbb{D}$ et toute donnée $t \in \mathbb{D}$ on a

$$[\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!](t) = \mathcal{F}(\mathbf{b}, \mathbf{v}, \mathbf{p}, t) \tag{5.2}$$

Le comportement d'un virus Smith fait apparaître un programme $\mathbf{b}$ calculant sa fonction de propagation, son code viral $\mathbf{v}$, le programme hôte $\mathbf{p}$ et l'état du système t. La fonction $\mathcal{F}$ spécifie ainsi comment propager l'infection en appliquant la procédure $\mathbf{b}$ aux éléments du système extrait de t.

La construction construction d'un virus Smith revient à résoudre le système d'équation suivant.

$$\begin{cases} [\![\mathbf{v}]\!](\mathbf{p}, t) = [\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!](t) \\ [\![\mathbf{v}]\!](\mathbf{p}, t) = \mathcal{F}(\mathbf{b}, \mathbf{v}, \mathbf{p}, t) \text{ avec } [\![\mathbf{b}]\!] = \mathcal{B} \end{cases}$$

De manière triviale, si l'on fixe $\mathcal{B} = \mathit{spec}$ et $\mathbf{b} = \mathbf{spec}$ alors on retrouve la construction d'un virus blueprint. Néanmoins, il est aussi possible exhiber d'autres solutions moins triviales en construisant une fonction de propagation $\mathcal{B}$ par un théorème de récursion. Nous résolvons alors ce système d'équations par un théorème de récursion double.

Théorème de récursion explicite. Les point fixes sont des programmes autoréférents dans le sens où il calculent une fonction en utilisant leur propre code. Par exemple dans l'équation suivante, le programme $\mathbf{v}$ calcule la fonction $\mathcal{F}$ en utilisant son propre code et la donnée t.

$$[\![\mathbf{v}]\!](t) = \mathcal{F}(\mathbf{v}, t)$$

La définition 6 fait apparaître une nouvelle propriété d'auto-référence : le programme **b** de l'équation (5.2) calcule la fonction $\mathcal{B}$ qui elle-même fournit un programme calculant une fonction en utilisant **b**. Nous sommes confrontés à un second niveau de récursion faisant intervenir une interprétation intermédiaire.

La construction d'un tel programme **b** donne lieu à un théorème de récursion peu commun dans la littérature. La forme la plus proche est le *théorème de récursion retardé* énoncé par Case dans [Cas74]. Une forme plus propre à notre utilisation est le *théorème de récursion explicite* énoncé dans [BKM06].

Théorème 7 (Théorème de récursion explicite)**.** *Pour tout programme* $\mathbf{r} \in \mathbb{D}$ *il existe un programme* $\mathbf{e} \in \mathbb{D}$ *tel que*

$$[\![[\![\mathbf{e}]\!](u)]\!](t) = [\![\mathbf{r}]\!](\mathbf{e}, u, t) \tag{5.3}$$

Démonstration. On note **xpl** le programme suivant où $x \in \mathbb{X}$ est une nouvelle variable.

$$\begin{array}{l} x = \mathbf{spec}(\mathbf{spec}, hd\,(work))\,; \\ work = \circ\,(x, tl\,(work)) \end{array} \tag{\textbf{xpl}}$$

On observe que pour toutes données $w, u, t \in \mathbb{D}$ le programme $\mathbf{q} = \mathbf{xpl}\,;\mathbf{r}$ satisfait

$$[\![\mathbf{q}]\!](w, u, t) = [\![\mathbf{r}]\!]\left(\mathit{spec}(\mathbf{spec}, w), u, t\right)$$

Par application du théorème de récursion au programme **q**, on obtient un programme $\mathbf{r}' \in \mathbb{D}$ tel que pour toutes données $u, t \in \mathbb{D}$

$$[\![\mathbf{r}']\!](u, t) = [\![\mathbf{q}]\!](\mathbf{r}', u, t) = [\![\mathbf{r}]\!]\left(\mathit{spec}(\mathbf{spec}, \mathbf{r}'), u, t\right)$$

On pose $\mathbf{e} = \mathit{spec}(\mathbf{spec}, \mathbf{r}')$ et on observe que $[\![\mathbf{e}]\!](u) = [\![\mathit{spec}(\mathbf{spec}, \mathbf{r}')]\!](u) = \mathit{spec}(\mathbf{r}', u)$. Il suit que pour toutes données $u, t \in \mathbb{D}$ on a

$$\begin{aligned} [\![[\![\mathbf{e}]\!](u)]\!](t) &= [\![\mathit{spec}(\mathbf{r}', u)]\!](t) \\ &= [\![\mathbf{r}']\!](u, t) \\ &= [\![\mathbf{r}]\!]\left(\mathit{spec}(\mathbf{spec}, \mathbf{r}'), u, t\right) \\ &= [\![\mathbf{r}]\!](\mathbf{e}, u, t) \end{aligned}$$

On conclut que le programme **e** satisfait le théorème. □

Dans la démonstration précédente, on notera que la construction de **e** est uniforme en **r**. Avec **k** le programme de la démonstration du théorème 2, le point fixe **e** du théorème de récursion explicite peut être obtenue par compilation du programme **r**.

$$\mathbf{e} = \mathit{spec}(\mathbf{spec}, \mathit{spec}(\mathbf{r}', \mathbf{r}')) \qquad \text{avec } \mathbf{r}' = \mathbf{k}\,;\mathbf{xpl}\,;\mathbf{r}$$

Cette remarque nous permettra de construire un compilateur de virus Smith.

Compilateur de virus Smith. La définition d'un virus Smith fait apparaître une double propriété d'auto-référence matérialisée par les programmes **b** et **v** dans l'équation (5.2). Depuis cette observation on peut soupçonner que la construction d'un virus Smith fait intervenir deux applications du théorème de récursion.

Propriété 8. *Pour tout programme* $\mathbf{r} \in \mathbb{D}$ *il existe un virus Smith relativement au comportement* $[\![\mathbf{r}]\!]$.

Démonstration. On note **wrap** le programme suivant où $x, y, z \in \mathbb{X}$ sont des nouvelles variables.

$$\begin{array}{l} [x, y, z] = work\ ; \\ work = \circ\,(x, hd\,(y), tl\,(y), z)\ ; \end{array} \qquad (\mathbf{wrap})$$

On observe que pour tous $w, u, \mathbf{p}, t \in \mathbb{D}$ le programme **wrap** ; **r** satisfait

$$[\![\mathbf{wrap}\ ;\ \mathbf{r}]\!]\,(w, \circ(u, \mathbf{p}), t) = [\![\mathbf{r}]\!]\,(w, u, \mathbf{p}, t)$$

On applique le théorème de récursion explicite au programme **wrap** ; **r**. On obtient un programme **b** et une fonction $\mathcal{B} = [\![\mathbf{b}]\!]$ tels que pour tous $u, \mathbf{p}, t \in \mathbb{D}$

$$[\![\mathcal{B}(u, \mathbf{p})]\!]\,(t) = [\![\mathbf{wrap}\ ;\ \mathbf{r}]\!]\,(\mathbf{b}, \circ(u, \mathbf{p}), t) = [\![\mathbf{r}]\!]\,(\mathbf{b}, u, \mathbf{p}, t)$$

Il reste à définir le code viral. Pour cela on applique le théorème de récursion au programme $\mathit{spec}(\mathbf{r}, \mathbf{b})$ et on obtient un programme **v** tel que pour tous $\mathbf{p}, t \in \mathbb{D}$

$$[\![\mathbf{v}]\!]\,(\mathbf{p}, t) = \left[\!\left[\mathit{spec}(\mathbf{r}, \mathbf{b})\right]\!\right]\,(\mathbf{v}, \mathbf{p}, t) = [\![\mathbf{r}]\!]\,(\mathbf{b}, \mathbf{v}, \mathbf{p}, t)$$

On conclut que **v** et $\mathcal{B}$ constituent un virus Smith relativement au comportement $[\![\mathbf{r}]\!]$. □

Exemple de virus Smith Grâce à la propriété précédente nous pouvons compiler toute spécification de virus Smith. Pour illustrer cela nous revenons sur l'exemple du virus parasite. On considère la spécification suivante notée **r**.

```
[x_b, x_v, x_p, x_q] = work ;   // décompose l'entrée
work = • ;                      // réinitialise l'état
foreach (y in x_q){             // pour chacun des programmes du système
 z = univ(x_b, x_v, y)          // calcule la forme infectée
 work = ∘ (z, work) ;           // et l'ajoute à la sortie
} ;
work = univ(p, z)               // exécute l'hôte original
```

On observe qu'un virus Smith relativement au comportement $[\![\mathbf{r}]\!]$ satisfait pour tous programmes $\mathbf{p}, \mathbf{q}_1, \cdots, \mathbf{q}_n \in \mathbb{D}$

$$\begin{aligned} [\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!]\,(\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet) &= [\![\mathbf{r}]\!]\,([\![\mathbf{b}]\!]\,(\mathbf{v}, \mathbf{q}_n), \cdots, [\![\mathbf{b}]\!]\,(\mathbf{v}, \mathbf{q}_1), \bullet) \\ &= [\![\mathbf{r}]\!]\,(\mathcal{B}(\mathbf{v}, \mathbf{q}_n), \cdots, \mathcal{B}(\mathbf{v}, \mathbf{q}_1), \bullet) \end{aligned}$$

Ce qui correspond bien au comportement d'un virus parasite.

En reprenant la preuve de la propriété 8, on peut obtenir un tel virus par compilation de la spécification $\mathbf{r}$.

$$\begin{array}{llll} \mathbf{v} = \mathit{spec}(\mathbf{r}'', \mathbf{r}'') & \text{avec} & \mathbf{r}'' = \mathbf{k}\,;\,\mathit{spec}(\mathbf{r}, \mathbf{b}) \\ \mathbf{b} = \mathit{spec}(\mathbf{spec}, \mathit{spec}(\mathbf{r}', \mathbf{r}')) & \text{avec} & \mathbf{r}' = \mathbf{k}\,;\,\mathbf{xpl}\,;\,\mathbf{wrap}\,;\,\mathbf{q} \end{array}$$

5.4 Retour sur le formalisme d'Adleman

Conservation du formalisme d'Adleman. Nous montrons que pour tout virus d'Adleman, il existe un virus au sens de la définition 3 ayant le même comportement. De plus nous verrons que cette correspondance est canonique. Soit $\mathcal{A} : \mathbb{D} \to \mathbb{D}$ un virus d'Adleman effectif. On prend un programme $\mathbf{a} \in \mathbb{D}$ calculant $\mathcal{A}$. On pose le code viral suivant noté $\mathbf{v}$ où $x, y \in \mathbb{X}$ sont des nouvelles variables.

$$\begin{array}{l} [x, y] = work\,; \\ x = \mathbf{a}(x)\,; \\ work = \mathbf{univ}(x, y) \end{array} \qquad (\mathbf{v})$$

Pour tout programme $\mathbf{p} \in \mathbb{D}$ et toute donnée $t \in \mathbb{D}$, on observe que ce code viral satisfait $[\![\mathbf{v}]\!](\mathbf{p}, t) = [\![\mathcal{A}(\mathbf{p})]\!](t)$. Pour être en accord avec le formalisme d'Adleman on prend pour fonction de propagation $\mathcal{B}(\mathbf{v}, \mathbf{p}) = \mathcal{A}(\mathbf{p})$. En effet $\mathcal{A}(\mathbf{p})$ est la forme infectée de $\mathbf{p}$ dans le formalisme d'Adleman. Et suivant notre définition, le programme $\mathcal{B}(\mathbf{v}, \mathbf{p})$ représente aussi la forme infectée de $\mathbf{p}$. On conclut en observant que le programme $\mathbf{v}$ est bien un virus relativement à la fonction de propagation $\mathcal{B}$: pour tout programme $\mathbf{p} \in \mathbb{D}$ et toute donnée $t \in \mathbb{D}$ on a

$$[\![\mathbf{v}]\!](\mathbf{p}, t) = [\![\mathcal{A}(\mathbf{p})]\!](t) = [\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!](t)$$

Conservation du formalisme de Cohen. Nous clamons que la définition que nous avons adoptée englobe aussi le formalisme de Cohen. La démonstration de cette propriété repose essentiellement sur l'encodage d'une machine de Turing. Ce travail étant fastidieux sans être d'un grand intérêt, nous ne présenterons pas cette démonstration ici. Néanmoins nous renvoyons le lecteur vers [Kac05] où une telle démonstration est réalisée.

Au delà du formalisme d'Adleman. Notre formalisme permet de modéliser des scénarios d'infection inaccessibles dans celui d'Adleman. Nous avons déjà souligné que les virus d'Adleman ne se propagent que par post-infection. Avec l'exemple du virus parasite de la partie 5.1 nous avons constaté que les pré-infections ne semblent pas poser problème dans notre formalisme. Nous allons plus loin en donnant un exemple de virus utilisant un mécanisme de post-infection mais qui ne rentre pas

dans les critères d'Adleman. On prend une fonction $\mathcal{A} : \mathbb{D} \to \mathbb{D}$ telle que pour tout programme $\mathbf{p}, \mathbf{q}_1, \cdots, \mathbf{q}_n \in \mathbb{D}$ on a

$$[\![\mathcal{A}(\mathbf{p})]\!]\,(\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet) = \circ(\mathcal{A}(\mathbf{q}'_m), \cdots, \mathcal{A}(\mathbf{q}'_1), \bullet)$$
$$\text{où } \circ\,(\mathbf{q}'_1, \cdots, \mathbf{q}'_m, \bullet) = [\![\mathbf{p}]\!]\,(\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet)$$

On observe qu'une forme infectée $\mathcal{A}(\mathbf{p})$ exécute le programme hôte $\mathbf{p}$ puis propage l'infection au travers des programmes résultants, tout en inversant leur ordre. Cette infection est tout à fait analogue au virus parasite de la partie 5.1, à la seule différence qu'ici nous employons une post-infection.

Nous montrons que $\mathcal{A}$ n'est pas un virus d'Adleman. On reprend les programmes **id** et **copy** définis précédemment. On rappelle que pour toutes données $t, u \in \mathbb{D}$ on a $[\![\mathbf{id}]\!]\,(t) = t$ et $[\![\mathbf{copy}]\!]\,(u, t) = \circ(u, u, t)$. On observe qu'aucun des comportements énoncés par Adleman n'opère sur l'entrée $\circ(\mathbf{id}, \mathbf{copy}, \bullet)$.

Nuisance. D'après la définition d'Adleman, on a les équations suivantes.

$$[\![\mathcal{A}(\mathbf{id})]\!]\,(\mathbf{id}, \mathbf{copy}, \bullet) = \circ(\mathbf{id}, \mathbf{copy}, \bullet)$$
$$[\![\mathcal{A}(\mathbf{copy})]\!]\,(\mathbf{id}, \mathbf{copy}, \bullet) = \circ(\mathbf{id}, \mathbf{id}, \mathbf{copy}, \bullet)$$

Il suit que $[\![\mathcal{A}(\mathbf{id})]\!]\,(\mathbf{id}, \mathbf{copy}, \bullet) \neq [\![\mathcal{A}(\mathbf{copy})]\!]\,(\mathbf{id}, \mathbf{copy}, \bullet)$. Il n'y a pas nuisance.

Infection. Par l'absurde on suppose que l'égalité suivante est satisfaite.

$$[\![\mathcal{A}(\mathbf{id})]\!]\,(\mathbf{id}, \mathbf{copy}, \bullet) = \circ(\mathcal{A}(\mathbf{id}), \mathcal{A}(\mathbf{copy}), \bullet)$$

Par définition de $\mathcal{A}$, il suit que $\mathcal{A}(\mathbf{id}) = \mathcal{A}(\mathbf{copy})$. Ce qui est absurde puisque dans le cas précédent, nous avons vu que ces programmes calculent des fonctions distinctes. Il n'y a pas infection.

Furtivité. Par l'absurde on suppose $[\![\mathcal{A}(\mathbf{id})]\!]\,(\mathbf{id}, \mathbf{copy}, \bullet) = \circ(\mathbf{id}, \mathbf{copy}, \bullet)$. Par définition de $\mathcal{A}$, il suit que $\mathcal{A}(\mathbf{copy}) = \mathbf{id}$. Ce qui est absurde puisque ces programmes calculent des fonctions distinctes :

$$[\![\mathbf{id}]\!]\,(\mathbf{id}, \mathbf{copy}) = \circ(\mathbf{id}, \mathbf{copy}, \bullet)$$
$$[\![\mathcal{A}(\mathbf{copy})]\!]\,(\mathbf{id}, \mathbf{copy}) = \circ(\mathcal{A}(\mathbf{id}), \mathcal{A}(\mathbf{id}), \mathcal{A}(\mathbf{copy}), \bullet)$$

Il n'y a pas furtivité.

On comprend que le scénario imposé par Adleman est très contraignant. Par contre cette infection ne pose aucun problème de modélisation dans notre formalisme. Comme auparavant, on prend un programme $\mathbf{v} \in \mathbb{D}$ satisfaisant $[\![\mathbf{v}]\!]\,(\mathbf{p}, t) = [\![\mathcal{A}(\mathbf{p})]\!]\,(t)$. On définit la fonction de propagation $\mathcal{B}(\mathbf{v}, \mathbf{p}) = \mathcal{A}(\mathbf{p})$. Et on observe que $\mathbf{v}$ est un virus relativement à la fonction de propagation $\mathcal{B}$. De plus $\mathcal{B}(\mathbf{v}, \mathbf{p})$ représente bien la forme infectées de l'hôte $\mathbf{p}$.

5.5 Conclusions

Fonctions de propagation. Dans ce chapitre nos avons toujours utilisé la fonction de spécialisation *spec* comme fonction de propagation canonique. Néanmoins, il est possible de construire d'autres fonctions de propagation. Comme dans l'exemple suivant.

On note **id** le programme $work = work$, on observe que ce programme calcule la *fonction identité*. Ensuite on définit la fonction de propagation $\mathcal{B} : \mathbb{D} \to \mathbb{D}$ qui à tous programmes $\mathbf{v}, \mathbf{p} \in \mathbb{D}$ associe le programme $\mathit{spec}(\mathbf{v}, \mathbf{id})\,;\,\mathbf{p}$. On pose $\mathbf{r}$ le programme suivant.

$$
\begin{array}{ll}
[x_{\mathbf{v}}, x_{\mathbf{p}}, x_{\mathbf{q}}] = work\,; & \text{// décompose l'entrée} \\
work = \mathbf{spec}(x_{\mathbf{v}}, \mathbf{id})\,; & \text{// construit le code à propager} \\
\mathit{foreach}\,(y\ \mathit{in}\ x_{\mathbf{q}})\{ & \\
\quad z = \circ\,(\circ(\mathbf{;}, work\,, y), z)\,; & \text{// infecte chacun des programmes du système} \\
\}\,; & \\
work = \mathbf{univ}(x_{\mathbf{p}}, z) & \text{// exécute l'hôte original}
\end{array}
\qquad (\mathbf{r})
$$

On rappel que **univ** est un auto-interpréteur, **spec** est un spécialiseur et *spec* est la fonction de spécialisation calculée par **spec**. C'est trois objets on été définis au chapitre 2.

On observe que pour tous programmes $\mathbf{v}, \mathbf{p}, \mathbf{q}_1, \cdots, \mathbf{q}_n \in \mathbb{D}$ le programme $\mathbf{r}$ satisfait

$$[\![\mathbf{r}]\!]\,(\mathbf{v}, \mathbf{p}, \circ(\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet)) = [\![\mathbf{p}]\!]\,(\mathcal{B}(\mathbf{v}, \mathbf{q}_n), \cdots, \mathcal{B}(\mathbf{v}, \mathbf{q}_1), \bullet)$$

Par application du théorème de récursion au programme $\mathbf{r}$ on obtient un programme $\mathbf{v} \in \mathbb{D}$ tel que

$$
\begin{aligned}
[\![\mathbf{v}]\!]\,(\mathbf{p}, \circ(\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet)) &= [\![\mathbf{r}]\!]\,(\mathbf{v}, \mathbf{p}, \circ(\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet)) \\
&= [\![\mathbf{p}]\!]\,(\mathcal{B}(\mathbf{v}, \mathbf{q}_n), \cdots, \mathcal{B}(\mathbf{v}, \mathbf{q}_1), \bullet)
\end{aligned}
$$

Par la dérivation suivante, on conclut que le code viral $\mathbf{v}$ et la fonction de propagation $\mathcal{B}$ satisfont l'équation du virus parasite.

$$
\begin{aligned}
[\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!]\,(\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet) &= [\![\mathit{spec}(\mathbf{v}, \mathbf{id})\,;\,\mathbf{p}]\!]\,(\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet) \\
&= [\![\mathbf{p}]\!]\,([\![\mathbf{r}]\!]\,(\mathbf{v}, \mathbf{id}, \circ(\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet))) \\
&= [\![\mathbf{p}]\!]\,([\![\mathbf{id}]\!]\,(\mathcal{B}(\mathbf{v}, \mathbf{q}_n), \cdots, \mathcal{B}(\mathbf{v}, \mathbf{q}_1), \bullet)) \\
&= [\![\mathbf{p}]\!]\,(\mathcal{B}(\mathbf{v}, \mathbf{q}_n), \cdots, \mathcal{B}(\mathbf{v}, \mathbf{q}_1), \bullet)
\end{aligned}
$$

On observe que $\mathbf{v}$ est un virus parasite dont la fonction de propagation est différente de la fonction *spec*. De plus, on remaque que cette fonction de propgagation $\mathcal{B}$ a été fixé avant de construire le virus. Ceci laisse présager la possibilité de construire des virus depuis une spécification non seulement du comportement mais aussi du mécanisme de propagation. Néanmois ce type de constructions ne semble pas trivial, comme l'illustre l'exemple précédent.

Perspectives de raffinement Notre formalisme est très souple, la contrepartie de cette généralité est que nous nommons virus des programmes qui intuitivement n'en sont pas. De manière générale, tout programme est un virus relativement à la fonction de spécialisation *spec*. On se rend compte que la notion de virus informatique repose sur la relation entre le code viral et la fonction de propagation : si on considère une fonction de propagation triviale alors on obtient une notion de virus triviale. Par conséquent, il est important de ne considérer que des fonctions de propagation qui sont réellement des vecteurs d'infection. La caractérisation d'une notion de fonction de propagation raisonnable est une piste de recherche très intéressante mais aussi très ardue. Une classification des comportements analogue à celle établie par Adleman pourrait permettre l'identification de classe de fonctions de propagation. L'auteur avoue s'être longuement penché sur cette problématique mais sans résultat notable.

Modélisation. Les virus blueprint et Smith sont deux outils de modélisation des comportement viraux. Le premier se focalise sur la duplication d'un code viral et le second sur la propagation d'une infection. Il faut garder à l'esprit que ces mécanismes restent des modèles et donc qu'il n'offre qu'une vision du mécanisme viral employé. En particulier le lecteur attentif aura remarqué que tout virus blueprint $\mathbf{v}$ de comportement $\mathcal{F}$ peut être vu comme un virus Smith de comportement $\mathcal{G}(\mathbf{b}, \mathbf{v}, \mathbf{p}, t) = \mathcal{F}(\mathbf{v}, \mathbf{p}, t)$. Réciproquement, un virus Smith de comportement $\mathcal{G}$ peut être vu comme un virus blueprint de $\mathcal{F}(\mathbf{v}, t) = \mathcal{G}(\mathbf{b}, \mathbf{v}, t)$.

Ces deux visions d'une même infection virale permettent de comprendre les différents mécanismes viraux. Prenons un exemple pour illustrer l'utilité de tels modèles. Supposons que nous soyons confronté à un virus Smith composé d'un code viral $\mathbf{v}$ et d'un programme $\mathbf{b}$ calculant sa fonction de propagation. Nous voulons identifier une bonne signature, c'est-à-dire un élément permettant de discriminer ce virus des programmes sains. Si le programme $\mathbf{b}$ emploie une faille de sécurité, nous pouvons construire notre signature sur $\mathbf{b}$, l'avantage étant que nous détecterons tout virus employant cette procédure de propagation. Par contre, si $\mathbf{b}$ est une procédure licite alors nous ne pouvons pas fonder la construction de la signature sur $\mathbf{b}$ car cela engendrerait de nombreux faux positif. Il faut donc se rabattre sur le code viral $\mathbf{v}$ et est y voir un mécanisme blueprint. Par exemple un algorithme libre de FTP[1] peut être considéré comme une procédure de propagation licite.

Compilateur de virus. Nous avons exhibé des compilateurs de virus fondés sur les théorèmes de récursion. On comprend que l'existence de virus informatiques est intimement reliée aux théorèmes de récursion. Cela nous permet de conclure qu'un système libre de tout virus ne satisfait pas le théorème de récursion. Cette piste sera développée aux chapitres 7 et 8 afin d'envisager des stratégies de protection.

[1] File Transfer Protocol

6 Mécanismes d'évolution

Afin de se prémunir contre la détection, certains virus informatiques sont capables de faire évoluer leur code ou leur fonctionnalité au cours de leur reproduction. On appel ces programmes des *virus polymorphes* car l'infection virale peut prendre plusieurs formes. Ces différentes formes sont appelées *mutations*. Nous avons vu que le polymorphisme a été envisagé très tôt dans la discipline : que ce soit dans le formalisme de von Neumann ou celui de Cohen, le phénomène de mutation est déjà présent.

Un virus polymorphe est plus difficile à détecter car là où il suffisait de rechercher un virus, il est ici nécessaire de rechercher toutes ses mutations. A l'heure actuelle, la plupart des attaques par infection informatique emploie des technologies de polymorphisme. Ces techniques sont toujours plus complexes et plus performantes, elles poussent parfois les logiciels antiviraux dans leurs retranchements. En ce sens le polymorphisme est un aspect majeur de la virologie informatique.

Nous complétons le travail réalisé au chapitre précédent en présentant des versions polymorphes des mécanismes blueprint et Smith. Cette étude se fera en deux temps. Tout d'abord, nous nous intéressons aux mutations syntaxiques : elles modifient la forme d'un virus sans altérer son comportement. Puis nous passons à un concept plus avancé où le comportement d'un virus polymorphe peut évoluer d'une génération à l'autre. Dans cette seconde partie, nous renforçons les liens entre théorèmes de récursion et constructions virales en présentant des méthodes de compilation de virus polymorphes analogues à celles du chapitre précédent.

6.1 Mutations syntaxiques

Moteurs de mutation. Un moteur de mutation est un programme qui permet de modifier les codes viraux au fur et à mesure de leur propagation tout en conservant le comportement. D'un point de vue abstrait, un *moteur de mutation* $\mathbf{mut} \in \mathbb{D}$ est un transformateur de programme qui à partir d'une entrée composée d'un programme $\mathbf{p} \in \mathbb{D}$ et d'une clé $i \in \mathbb{D}$ retourne un programme $\mathbf{p}_i = [\![\mathbf{mut}]\!](\mathbf{p}, i)$ syntaxiquement différent de $\mathbf{p}$ mais calculant la même fonction.

$$\mathbf{p}_i \neq \mathbf{p} \qquad\qquad [\![\mathbf{p}_i]\!] = [\![\mathbf{p}]\!]$$

Habituellement, un moteur de mutation est aussi injectif en la clé i, c'est à dire $i \neq j$ implique $[\![\mathbf{mut}]\!](\mathbf{p}, i) \neq [\![\mathbf{mut}]\!](\mathbf{p}, j)$. Ainsi deux mutations d'un même programme par des clés distinctes engendrent des programmes différents.

Fonctions de padding. En pratique, les moteurs de mutation sont construits sur des procédures de chiffrement ou des mécanismes de substitution de commande. Du point de vue de la calculabilité, un tel programme calcule simplement une fonction de padding comme définie par Rogers [Rog67].

Définition 9 (Fonctions de padding)**.** Une fonction totale pad est une *fonction de padding* si pour tout programme $\mathbf{p} \in \mathbb{D}$ et toutes données $i, j \in \mathbb{D}$ elle satisfait les équations suivantes.

$$[\![pad\,(\mathbf{p}, i)]\!] = [\![\mathbf{p}]\!] \qquad\qquad i \neq j \Rightarrow pad\,(\mathbf{p}, i) \neq pad\,(\mathbf{p}, i)$$

Une fonction de padding est effective si elle est calculable.

D'après le *lemme de padding* [Rog67], il existe une fonction de padding effective dans tout langage de programmation acceptable. Ce lemme est souvent déprécié à un simple ajout de commandes inutiles. Dans le contexte de la virologie informatique, il prend une autre dimension puisqu'il assure l'existence de moteurs de mutation, quelque soit le modèle de calcul raisonnable considéré. A ce propos, nous invitons le lecteur à consulter la démonstration du théorème d'isomorphisme de Rogers [Rog58] où ce lemme est démontré de manière non triviale, c'est à dire sans utiliser l'argument des commandes inutiles. En particulier, cette démonstration utilise le théorème de récursion.

Une étude détaillée de ce lemme permet d'obtenir un résultat dont les implications pratiques sont d'actualité. On se munit d'un virus blueprint représenté par un code viral $\mathbf{v} \in \mathbb{D}$. De plus, on se donne un programme $\mathbf{av} \in \mathbb{D}$ calculant une fonction totale et satisfaisant l'équation suivante pour tout programme $\mathbf{p} \in \mathbb{D}$.

$$\big([\![\mathbf{av}]\!]\,(\mathbf{p}) = \circ(\bullet, \bullet)\big) \Rightarrow [\![\mathbf{p}]\!] = [\![\mathbf{v}]\!]$$

Ce programme représente un logiciel antiviral qui détecte un ensemble de programmes calculant la même fonction que le code viral $\mathbf{v}$. En d'autres termes, ce détecteur ne produit pas de faux positif. Nous montrons qu'il existe un programme $\mathbf{v}' \in \mathbb{D}$ tel que $[\![\mathbf{av}]\!]\,(\mathbf{v}') \neq \circ(\bullet, \bullet)$ et $[\![\mathbf{v}']\!] = [\![\mathbf{v}]\!]$. C'est-à-dire que $\mathbf{v}'$ est une mutation du virus $\mathbf{v}$ non détectée par $\mathbf{av}$.

Propriété 10. *Soient* $\mathbf{av}, \mathbf{v} \in \mathbb{D}$ *deux programmes tels que pour tout programme* $\mathbf{p} \in \mathbb{D}$ *on a* $\big([\![\mathbf{av}]\!]\,(\mathbf{p}) = \circ(\bullet, \bullet)\big) \Rightarrow [\![\mathbf{p}]\!] = [\![\mathbf{v}]\!]$*. Si* $\mathbf{av}$ *calcule une fonction totale alors il existe un programme* $\mathbf{v} \in \mathbb{D}$ *tel que* $[\![\mathbf{av}]\!]\,(\mathbf{v}') \neq \circ(\bullet, \bullet)$ *et* $[\![\mathbf{v}']\!] = [\![\mathbf{v}]\!]$*.*

Démonstration. On prend $\mathbf{sane} \in \mathbb{D}$ un programme calculant une fonction distincte de celle calculée par $\mathbf{v}$. Par hypothèse on a $[\![\mathbf{av}]\!]\,(\mathbf{sane}) \neq \circ(\bullet, \bullet)$. On note $\mathbf{r}$ le programme suivant.

$$\begin{array}{l} x = \mathbf{av}(work)\,; \\ if\,(x == \circ\,(\bullet, \bullet)) \\ \{\ work = \mathbf{sane}\ \} \\ else \\ \{\ work = \mathbf{v}\ \} \end{array} \qquad (\mathbf{r})$$

On observe que pour tout programme $\mathbf{p} \in \mathbb{D}$, si $[\![\mathbf{av}]\!](\mathbf{p}) = \circ(\bullet, \bullet)$ alors $[\![\mathbf{r}]\!](\mathbf{p}) = \mathbf{sane}$, sinon $[\![\mathbf{r}]\!](\mathbf{p}) = \mathbf{v}$. On applique le théorème de récursion au programme suivant

$$\begin{array}{l} [x, y] = work\ ; \\ z = \mathbf{r}(x)\ ; \\ work = \mathbf{univ}(z) \end{array}$$

On obtient un programme $\mathbf{v}' \in \mathbb{D}$ tel que $[\![\mathbf{v}']\!](t) = \mathit{univ}([\![\mathbf{r}]\!](\mathbf{v}'), t)$. Par l'absurde, supposons que $[\![\mathbf{av}]\!](\mathbf{v}') = \circ(\bullet, \bullet)$. Alors $[\![\mathbf{r}]\!](t) = \mathbf{sane}$ et par conséquent $[\![\mathbf{v}']\!] = [\![\mathbf{sane}]\!]$ ce qui est absurde par définition de $\mathbf{av}$. On conclut que $\mathbf{v}'$ satisfait la propriété. □

D'un point de vue pratique, cette démonstration nous montre qu'une analyse de type boîte noire d'un logiciel antiviral peut permettre de créer des code viraux non détecté. Une autre façon de présenter ce résultat est d'invoquer le théorème de Rice : il stipule que tout ensemble extentionel[1] décidable est soit le domaine entier, soit l'ensemble vide.

Ce résultat n'est pas directement applicable en pratique car les logiciels antiviraux produisent des faux positifs, c'est à dire qu'un programme détecté ne se comporte pas forcément comme un virus. Néanmoins les articles [CJ04, Fil06b] montrent l'effectivité de cette méthode de mutation avec des virus réels, et ce même dans le cadre d'une détection produisant des faux positifs.

Mutation de virus blueprint. Etant donné une spécification $\mathbf{r} \in \mathbb{D}$, on rappelle qu'un virus blueprint de comportement $[\![\mathbf{r}]\!]$ est un programme $\mathbf{v} \in \mathbb{D}$ tel que pour toute donnée $t \in \mathbb{D}$ on a $[\![\mathbf{v}]\!](t) = [\![\mathbf{r}]\!](\mathbf{v}, t)$. Par application directe du théorème de récursion, on sait construire de manière systématique un tel virus blueprint quelque soit la spécification $\mathbf{r}$.

Nous voulons obtenir un virus blueprint polymorphe depuis une spécification $\mathbf{r}$. Cela revient à générer un ensemble de code viraux $\mathbb{V} \subset \mathbb{D}$ tel que pour tout élément $\mathbf{v} \in \mathbb{V}$, il y a un second élément $\mathbf{v}' \in \mathbb{V}$ satisfaisant l'équation des virus blueprint énoncée au chapitre 5. C'est à dire que pour toute donnée $t \in \mathbb{D}$ on a

$$[\![\mathbf{v}]\!](t) = [\![\mathbf{r}]\!](\mathbf{v}', t) \tag{6.1}$$

Pour cela, on se muni d'un moteur de mutation $\mathbf{mut} \in \mathbb{D}$ et on pose $\mathbf{r}'$ le programme suivant où $x \in \mathbb{X}$ est une nouvelle variable.

$$\begin{array}{ll} [x, y] = work\ ; & \\ z = \mathbf{mut}(x, y)\ ; & (\mathbf{r}') \\ work = \mathbf{r}(z, x) & \end{array}$$

[1] Un ensemble *extentionel* $\mathbb{A}$ est tel que s'il existe un programme calculant la même fonction que l'un des programmes de $\mathbb{A}$ alors ce premier programme est un élément de $\mathbb{A}$.

On observe que pour toutes données $t, u \in \mathbb{D}$ le programme $\mathbf{r}'$ satisfait

$$[\![\mathbf{r}']\!]\,(u,t) = [\![\mathbf{r}]\!]\,([\![\mathbf{mut}]\!]\,(u,t)\,,t)$$

Par application du théorème de récursion, on obtient un programme $\mathbf{v} \in \mathbb{D}$ tel que pour toute donnée $t \in \mathbb{D}$ on a

$$[\![\mathbf{v}]\!]\,(t) = [\![\mathbf{r}']\!]\,(\mathbf{v},t) = [\![\mathbf{r}]\!]\,(\mathbf{v}_t,t) \qquad \text{avec } \mathbf{v}_t = [\![\mathbf{mut}]\!]\,(\mathbf{v},t)$$

On rappelle que pour toute donnée $i \in \mathbb{D}$, les programmes $\mathbf{v}$ et $[\![\mathbf{mut}]\!]\,(\mathbf{v},i)$ calculent la même fonction. Par conséquent, pour toutes données $t, i \in \mathbb{D}$ on a

$$[\![\mathbf{v}_i]\!]\,(t) = [\![\mathbf{r}]\!]\,(\mathbf{v}_t,t) \qquad \text{avec} \quad \begin{array}{l}\mathbf{v}_i = [\![\mathbf{mut}]\!]\,(\mathbf{v},i)\\ \mathbf{v}_t = [\![\mathbf{mut}]\!]\,(\mathbf{v},t)\end{array}$$

Intuitivement, toute mutation $\mathbf{v}_i$ exécutée depuis l'état du système t propage le code viral $\mathbf{v}_t$. On a bien un virus blueprint polymorphe dans le sens ou le code viral propagé évolue lors de la reproduction.

Mutation de virus Smith. De manière analogue à la mutation de virus blueprint, on peut modifier la fonction d'infection au cours de la reproduction. Il suffit d'appliquer un moteur de mutation aux formes infectées.

D'un point de vue formel cela revient à construire un code viral $\mathbf{v} \in \mathbb{D}$ et à générer un ensemble de fonction de propagation $\mathbb{B} \subset (\mathbb{D} \to \mathbb{D})$ tel que pour toute fonction $\mathcal{B} \in \mathbb{B}$ il y a une seconde fonction $\mathcal{B}' \in \mathbb{B}$ satisfaisant le système d'équations des virus Smith énoncée au chapitre 5. C'est à dire que pour tout programme $\mathbf{p} \in \mathbb{D}$ et toute donnée $t \in \mathbb{D}$ on a

$$\begin{cases} [\![\mathbf{v}]\!]\,(\mathbf{p},t) = [\![\mathcal{B}(\mathbf{v},\mathbf{p})]\!]\,(t) \\ [\![\mathbf{v}]\!]\,(\mathbf{p},t) = \mathcal{F}\,(\mathbf{b}',\mathbf{v},\mathbf{p},t) \text{ avec } [\![\mathbf{b}']\!] = \mathcal{B}' \end{cases} \tag{6.2}$$

Nous construisons un tel ensemble de solutions par l'intermédiarte du moteur de mutation **mut**. On note $\mathbf{mut}'$ le programme suivant.

$$work = \mathbf{mut}(work\,,work) \qquad (\mathbf{mut}')$$

Etant donné un comportement $\mathcal{F} : \mathbb{D} \to \mathbb{D}$, il suffit de construire un virus Smith pour le comportement $\mathcal{G}(\mathbf{b},\mathbf{v},\mathbf{p},t) = \mathcal{F}\,(\mathbf{mut}'\,;\mathbf{b},\mathbf{v},\mathbf{p},t)$. On obtient un code viral $\mathbf{v} \in \mathbb{D}$ et une fonction de propagation $\mathcal{B} : \mathbb{D} \to \mathbb{D}$ tels que pour tout programme $\mathbf{p} \in \mathbb{D}$ et toute donnée $t \in \mathbb{D}$ on a

$$\begin{aligned}[\![\mathbf{v}]\!]\,(\mathbf{p},t) = [\![\mathcal{B}(\mathbf{v},\mathbf{p})]\!]\,(t) &= \mathcal{G}(\mathbf{b},\mathbf{v},\mathbf{p},t)\\ &= \mathcal{F}\,(\mathbf{mut}'\,;\mathbf{b},\mathbf{v},\mathbf{p},t)\end{aligned}$$

Pour toute clé $i \in \mathbb{D}$ on note $\mathcal{B}_i(\mathbf{v},\mathbf{p}) = [\![\mathbf{mut}]\!]\,(\mathcal{B}_i(\mathbf{v},\mathbf{p}),i)$, il suit

$$[\![\mathcal{B}_i(\mathbf{v},\mathbf{p})]\!]\,(t) = \mathcal{F}\,(\mathbf{mut}'\,;\mathbf{b},\mathbf{v},\mathbf{p},t)$$

où $\mathbf{mut}'\,;\mathbf{b}$ calcule la fonction de propagation $\mathcal{B}_{\mathcal{B}(\mathbf{v},\mathbf{q})}$ avec $\mathbf{v}$ le code viral propagé et $\mathbf{q}$ l'hôte infecté. On observe que les formes infectées sont mutées.

6.2 Mutation du comportement

Les constructions que nous avons présentées dans la section précédentes sont assez limitées. D'une part, elles dépendent d'un moteur de mutation fixé et d'autre part elles sont purement syntaxiques, c'est-à-dire que le comportement viral n'évolue pas au cours de la propagation. Nous étudions maintenant une notion plus poussée de polymorphisme. En résolvant des systèmes d'équations plus généraux que (6.1) et (6.2).

Distributions de virus. Une distribution de virus est constituée d'un ensemble de codes viraux se propageant par un même vecteur d'infection. Un tel objet permet de modéliser des virus polymorphes, chaque code viral étant vue comme une mutation.

Définition 11 (Distribution de virus)**.** Une fonction $\mathit{dst} : \mathbb{D} \to \mathbb{D}$ est une *distribution de virus* relativement à une fonction de propagation $\mathcal{B}$ si pour toute donnée $i \in \mathbb{D}$ le programme $\mathit{dst}(i)$ est un virus relativement à $\mathcal{B}$.
Pour toute distribution de virus dst, la donnée i est la *génération* du code viral $\mathit{dst}(i)$.
Une distribution de virus est effective si elle est calculable.

Distributions blueprint Une distribution blueprint est constituée d'un ensemble de virus blueprint. Ces codes viraux n'utilisent pas explicitement de fonction de propagation pour transmettre l'infection.

Définition 12 (Distributions blueprint)**.** Soit $\{\mathcal{F}_i\}_{i\in\mathbb{D}}$ un ensemble de fonctions de $\mathbb{D}$ vers $\mathbb{D}$. Une *distribution blueprint* relativement aux comportements $\{\mathcal{F}_i\}_{i\in\mathbb{D}}$ est un programme $\mathbf{dst} \in \mathbb{D}$ satisfaisant pour toute génération $i \in \mathbb{D}$

$$[\![\mathbf{v}_i]\!](t) = \mathcal{F}_i(\mathbf{dst}, t) \qquad \text{avec } \mathbf{v}_i = [\![\mathbf{dst}]\!](i) \tag{6.3}$$

Dans cette définition, le programme **dst** prend en entrée une génération i et retourne un code viral de cette génération.

L'équation (6.3) peut être vue comme une généralisation de l'équation (6.1). On résout cette équation par l'application du théorème de récursion explicite. On peut ainsi compiler une distributions blueprint à partir d'une spécification de ses comportements.

Propriété 13. *Soit $\{\mathcal{F}_i\}_{i\in\mathbb{D}}$ un ensemble de fonctions de $\mathbb{D}$ vers $\mathbb{D}$. Si la fonction $\mathcal{F}(u,i,t) = \mathcal{F}_i(u,t)$ est semi-calculable alors il existe une distribution blueprint relativement aux comportements $\{\mathcal{F}_i\}_{i\in\mathbb{D}}$.*

Démonstration. Soit $\mathbf{r} \in \mathbb{D}$ un programme calculant $\mathcal{F}$. On applique le théorème de récursion explicite à $\mathbf{r}$. On obtient un programme $\mathbf{dst} \in \mathbb{D}$ tel que pour toutes données $t, i \in \mathbb{D}$ on a

$$[\![\mathbf{v}_i]\!](t) = \mathcal{F}(\mathbf{dst}, i, t) = \mathcal{F}_i(\mathbf{dst}, t) \qquad \text{avec } \mathbf{v}_i = [\![\mathbf{dst}]\!](i)$$

On conclut que **dst** est une distribution blueprint satisfaisant la propriété. □

Exemple d'une distribution blueprint. On reprend l'exemple du virus écraseur du chapitre 5 avec la spécification **r** suivante.

$$
\begin{array}{ll}
[x_{\mathbf{dst}}, x_i, x_t] = work\,; & // \text{ décompose l'entrée} \\
work = \bullet\,; & // \text{ réinitialise l'état} \\
foreach\,(y\ in\ x_i)\{ & \\
\ z = \mathbf{univ}(x_{\mathbf{dst}}, y)\,; & // \text{ calcule le code viral de génération } y \\
\ work = \circ\,(z, work)\,; & // \text{ ajoute ce code à la sortie} \\
\}\,; &
\end{array}
$$

Par application du théorème de récursion explicite, on obtient un générateur de blueprint $\mathbf{dst} \in \mathbb{D}$ satisfaisant pour toute génération $i = \circ(i_1, \cdots, i_m, \bullet)$ et toute liste de programmes $\mathbf{q}_1, \cdots, \mathbf{q}_n \in \mathbb{D}$

$$[\![\mathbf{v}_i]\!]\,(\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet) = \circ(\mathbf{v}_{i_m}, \cdots, \mathbf{v}_{i_1}, \bullet) \qquad \text{avec } \forall u \in \mathbb{D} : \mathbf{v}_u = [\![\mathbf{dst}]\!]\,(u)$$

Dans cet exemple, un code viral de génération $i = \circ(i_1, \cdots, i_m, \bullet)$ remplace l'ensemble des programmes du système par m codes viraux de générations $i_1, \cdots, i_m$. Le comportement viral évolue dans le sens où plus la génération décroît moins on introduit de codes viraux sur le système.

On rappelle que le programme **dst** peut être compilé depuis la spécification **r** de la manière suivante.

$$\mathbf{dst} = \mathit{spec}(\mathbf{spec}, \mathit{spec}(\mathbf{r}', \mathbf{r}')) \qquad \text{avec } \mathbf{r}' = \mathbf{k}\,;\mathbf{xpl}\,;\mathbf{r}$$

Où **k** est le programme défini dans la preuve du théorème de récursion et **xpl** celui défini dans la preuve du théorème de récursion explicite.

Distributions Smith. On passe maintenant à la génération de solution pour le système d'équations Smith. Nous modélisons ces solutions par l'intermédiare de la notion de distribution Smith

Définition 14. Soit $\{\mathcal{F}_i\}_{i\in\mathbb{D}}$ un ensemble de fonctions de $\mathbb{D}$ vers $\mathbb{D}$. Une *distribution Smith* relativement aux comportements $\{\mathcal{F}_i\}_{i\in\mathbb{D}}$ est un couple de programmes $\mathbf{dst}, \mathbf{b} \in \mathbb{D}$ satisfaisant pour toute génération $i \in \mathbb{D}$

$$[\![\mathcal{B}(\mathbf{v}_i, \mathbf{p})]\!]\,(t) = \mathcal{F}_i(\mathbf{b}, \mathbf{dst}, \mathbf{p}, t) \qquad \text{avec } \begin{array}{l} [\![\mathbf{b}]\!] = \mathcal{B} \\ \mathbf{v}_i = [\![\mathbf{dst}]\!]\,(i) \end{array} \tag{6.4}$$

Comme dans le cas d'un virus Smith simple, le programme **b** calcule la fonction de propagation. Le programme **dst** calcule les codes viraux depuis leur génération. Chaque comportement $\mathcal{F}_i$ décrit comment propager l'infection en utilisant la procédure **b**, le générateur de codes viraux **dst**, le programme hôte **p** et l'état du système t.

Construire une distribution Smith revient à générer un ensemble de codes viraux $\{\mathbf{dst}(i) \mid i \in \mathbb{D}\}$ et à construire une fonction de propagation $\mathcal{B}$ qui satisfont le système d'équations suivant.

$$\begin{cases} [\![\mathbf{v}_i]\!]\,(t) = [\![\mathcal{B}(\mathbf{v}_i, \mathbf{p})]\!]\,(t) \\ [\![\mathbf{v}_i]\!]\,(t) = \mathcal{F}\,(\mathbf{b}, \mathbf{dst}, t) \\ \quad [\![\mathbf{b}]\!] = \mathcal{B} \\ \quad\quad \mathbf{v}_i = [\![\mathbf{dst}]\!]\,(i) \end{cases}$$

Pour résoudre ce système, il suffit d'appliquer le théorème de récursion explicite.

Propriété 15. *Soit $\{\mathcal{F}_i\}_{i\in\mathbb{D}}$ un ensemble de fonctions de $\mathbb{D}$ vers $\mathbb{D}$. Si la fonction $\mathcal{F}(w, u, i, t) = \mathcal{F}_i(w, u, t)$ est semi-calculable alors il existe une distribution Smith relativement aux comportements $\{\mathcal{F}_i\}_{i\in\mathbb{D}}$.*

Démonstration. Soit $\mathbf{r} \in \mathbb{D}$ un programme calculant $\mathcal{F}$. On applique le théorème de récursion explicite au programme $\mathit{spec}(\mathbf{r}, \mathbf{spec})$ et on obtient un programme $\mathbf{dst}$ tel que pour tous $i, \mathbf{p}, t \in \mathbb{D}$ on a

$$\begin{aligned} [\![\mathbf{v}_i]\!]\,(\mathbf{p}, t) &= \left[\!\left[\mathit{spec}(\mathbf{r}, \mathbf{spec})\right]\!\right](\mathbf{dst}, i, \mathbf{p}, t) && \text{avec } \mathbf{v}_i = [\![\mathbf{dst}]\!]\,(i) \\ &= \mathcal{F}\,(\mathbf{spec}, \mathbf{dst}, i, \mathbf{p}, t) \\ &= \mathcal{F}_i(\mathbf{spec}, \mathbf{dst}, \mathbf{p}, t) \end{aligned}$$

En posant $\mathbf{b} = \mathbf{spec}$ et $\mathcal{B} = \mathit{spec}$ on obtient

$$\begin{aligned} [\![\mathcal{B}(\mathbf{v}_i, \mathbf{p})]\!]\,(t) &= \left[\!\left[\mathit{spec}(\mathbf{v}_i, \mathbf{p})\right]\!\right](t) \\ &= [\![\mathbf{v}_i]\!]\,(\mathbf{p}, t) \\ &= \mathcal{F}_i(\mathbf{spec}, \mathbf{dst}, \mathbf{p}, t) \\ &= \mathcal{F}_i(\mathbf{b}, \mathbf{dst}, \mathbf{p}, t) && \text{avec } [\![\mathbf{b}]\!] = \mathcal{B} \end{aligned}$$

On conclut que les programmes $\mathbf{dst}$ et $\mathbf{b}$ constituent une distribution Smith satisfaisant la propriété. □

Exemple de distributions Smith. Nous donnons un exemple de compilation d'une distribution Smith en reprenant l'exemple du virus parasite. On considère la spécifi-

cation suivante notée **r**.

$[x_{\mathbf{b}}, x_{\mathbf{dst}}, x_i, x_{\mathbf{p}}, x_t] = work$; // décompose l'entrée
$work = \bullet$; // réinitialise l'état
$foreach\,(y\ in\ x_i)\{$
$z = \circ\,(hd\,(x_t), z)$; // conserve quelques programmes
} ;
$foreach\,(y\ in\ z)\{$ // pour chacun de ces programmes
$x_i = tl\,(x_i)$; // décrémente la génération du virus
$x_{\mathbf{v}} = \mathbf{univ}(x_{\mathbf{dst}}, x_i)$; // calcule un nouveau code viral
$\alpha = \mathbf{univ}(x_{\mathbf{b}}, x_{\mathbf{v}}, y)$; // infecte le programme
$work = \circ\,(\alpha, work)$; // ajoute le programme infecté à la sortie
} ;
$work = \mathbf{univ}(\mathbf{p}, work)$ // exécute l'hôte original

En appliquant la propriété 15 on obtient une distribution Smith satisfaisant pour toute génération $i = \circ(i_1, \cdots, i_m, \bullet)$, tout hôte $\mathbf{p} \in \mathbb{D}$ et tout état $t = \circ(\mathbf{q}_1, \cdots, \mathbf{q}_n, \bullet)$

$$\begin{aligned}
[\![\mathcal{B}(\mathbf{v}_i, \mathbf{p})]\!]\,(t) &= [\![\mathbf{r}]\!]\,(\mathbf{b}, \mathbf{dst}, i, \mathbf{p}, t) \quad \text{avec } [\![\mathbf{b}]\!] = \mathcal{B} \\
&= [\![\mathbf{p}]\!]\,(\mathcal{B}(\mathbf{v}_m, \mathbf{q}_m), \cdots, \mathcal{B}(\mathbf{v}_1, \mathbf{q}_1), \bullet) \\
\text{avec} \quad \mathbf{v}_i &= [\![\mathbf{dst}]\!]\,(i) \\
\mathbf{v}_1 &= [\![\mathbf{dst}]\!]\,(\bullet) \\
&\cdots \\
\mathbf{v}_m &= [\![\mathbf{dst}]\!]\,(i_2, \cdots, i_m, \bullet)
\end{aligned}$$

On observe qu'une mutation de génération $i = \circ(i_1, \cdots, i_m, \bullet)$ conserve les m premiers programmes du système et les infecte. Les autres programmes sont simplement supprimés. Le comportement de cette infection évolue dans le sens où plus la génération décroît, plus on élimine de programmes sur le système. Comme la génération décroît lors de la reproduction, à termes le système ne contiendra plus aucun programme.

Mutation de la fonction de propagation. En suivant le même schéma, il est possible de construire des distributions de virus Smith dont la fonction de propagation évolue au cours des reproductions. C'est à dire générer non seulement un ensemble de codes viraux mais aussi un ensemble de fonction de propagation qui satisfont le système d'équations Smith. On se référera à [BKM07] pour un développement détaillé.

6.3 Conclusions et compléments

Nous avons mis en évidence un lien entre les moteurs de mutation et les fonctions de padding dont les fondements ne sont pas encore étudiés. Etant muni d'un

moteur de mutation, nous avons exposé comment muter un code viral ou une fonction de propagation. Puis, en introduisant le concept de distribution, nous sommes passés à une notion de mutation plus complexe faisant intervenir une évolution du comportement. En reliant la construction de virus polymorphe à la résolution de systèmes d'équations, nous avons compilé des distributions blueprint et Smith par l'intermédaire des théorèmes de récursion présentés au chapitre 5. Ceci renforce encore le parallèle entre virus informatiques et théorèmes de récursion.

Méthodes de protection

7 Virologie et capacité de calcul

L'objectif de cette partie est de mener dans le contexte des langages de programmation un travail comparable à celui réalisé von Neumann [vN66] dans le domaine des automates cellulaires. Nous nous interrogeons sur les briques élémentaires suffisantes à la construction de programmes auto-reproducteurs, et par suite, à l'existence de virus informatiques.

Nous avons vue dans les chapitres précédents que cette problématique a été étudiée par Kleene [Kle52] : son théorème de récursion permet de construire des programmes auto-référents dans tout langage de programmation acceptable. En utilisant ce théorème comme support pour notre formalisme nous pouvons affirmer qu'il existe des virus informatiques de manière intrinsèque dans tout modèle de calcul raisonnable. Ici, nous allons plus loin en nous intéressant aux langages ne satisfaisant pas la notion d'acceptabilité.

Dans [Coh89] Cohen s'interroge sur l'existence de systèmes libres de toute construction virale. De manière pragmatique, il propose de limiter les capacités de calcul au point qu'il ne soit plus possible qu'un programme puisse s'auto-reproduire. Ce chapitre met en doute la faisabilité d'une telle solution.

Nous montrons qu'un sous-langage de `While`, aux capacités de calcul très limitées, possède l'ensemble des constructions virales établies précédemment. Nous ne nous focaliserons pas sur l'aspect négatif de ce résultat. Nous pensons plutôt que la dissection minutieuse des mécanismes d'auto-reproduction est essentielle à la compréhension des fondements de la virologie informatique. Ainsi, un regard attentionné aux résultats présentés peut mener vers d'autres solutions antivirales.

Nous suivons la démarche suivante. Dans un premier temps nous introduisons le langage `Ker` permettant d'étudier les mécanismes viraux dans un contexte de capacités de calcul limitées. Puis nous constatons que les constructions virales présentées aux chapitres précédents sont toujours valides. Enfin nous introduisons un nouveau mécanisme de reproduction par vecteur d'infection plus en accord avec les capacités du langage `Ker`. Enfin, nous observons que même si des virus existent, il ne semble pas toujours possible de les construire par les seules capacités de `Ker`. Cette dernière remarque ouvre des pistes de recherche pouvant mener à la conception de systèmes immunisés.

7.1 Le langage `Ker`

Syntaxe. Nous considérons un sous-langage de `While` nommé `Ker`. Il est limité aux opérations de ce couplage, de projection, d'assignation et de composition. Ces opérations étant fondamentales à un système raisonnable, il semble difficile d'envisager un langage non trivial aux capacités encore plus restreintes. La syntaxe du langage `Ker` est la suivante.

$$\begin{array}{rrcl}
\text{Domaine :} & \mathbb{D} & ::= & \bullet \mid \circ(\mathbb{D}, \mathbb{D}) \\
\text{Variables :} & \mathbb{X} & ::= & work \mid x \mid y \mid \cdots \\
\text{Expressions :} & \mathbb{E} & ::= & \mathbb{D} \mid \mathbb{X} \mid \circ(\mathbb{E}, \mathbb{E}) \mid hd(\mathbb{E}) \mid tl(\mathbb{E}) \\
\text{Commandes :} & \mathbb{C}^{\texttt{Ker}} & ::= & \mathbb{X} \texttt{=} \mathbb{E} \mid \mathbb{C}^{\texttt{Ker}} \texttt{;} \mathbb{C}^{\texttt{Ker}} \\
\text{Programmes :} & \mathbb{P}^{\texttt{Ker}} & ::= & \mathbb{C}^{\texttt{Ker}}
\end{array}$$

Sémantique. Comme pour le langage `While` un programme de Ker est une commande. Concernant la sémantique de `Ker`, elle préserve celle de `While`.

$$\begin{array}{rrcl}
\text{Domaine :} & (\!|t|\!)(\sigma) & = & t \\
\text{Variables :} & (\!|x|\!)(\sigma) & = & \sigma(x) \\
\text{Expressions :} & (\!|\circ(e_1, e_2)|\!)(\sigma) & = & \circ((\!|e_1|\!)(\sigma), (\!|e_2|\!)(\sigma)) \\
& (\!|hd(e)|\!)(\sigma) & = & \begin{cases} \bullet & \text{si } (\!|e|\!)(\sigma) = \bullet \\ t_1 & \text{si } (\!|e|\!)(\sigma) = \circ(t_1, t_2) \end{cases} \\
& (\!|tl(e)|\!)(\sigma) & = & \begin{cases} \bullet & \text{si } (\!|e|\!)(\sigma) = \bullet \\ t_2 & \text{si } (\!|e|\!)(\sigma) = \circ(t_1, t_2) \end{cases} \\
\text{Commandes :} & [\![x \texttt{=} e]\!](\sigma) & = & \sigma[x \mapsto (\!|e|\!)(\sigma)] \\
& [\![c_1 \texttt{;} c_2]\!](\sigma) & = & [\![c_2]\!]([\![c_1]\!](\sigma)) \\
\text{Programmes :} & [\![\mathbf{p}]\!](t) & = & \sigma(work) \text{avec } \sigma = [\![\mathbf{p}]\!]([work \mapsto t])
\end{array}$$

Syntaxe concrète. On notera que $[\![\ ]\!]$ n'est pas ambiguë puisque d'une part tout programme de `Ker` est aussi un programme de `While`, et d'autre part la sémantique de `While` est préservée par `Ker`.

Nous conservons la syntaxe concrète des programmes de `While`.

$$\begin{array}{rrcl}
\text{Domaine :} & \underline{t} & = & \circ(\textbf{\textit{quote}}, t) \\
\text{Variables :} & \underline{x} & = & \circ(\textbf{\textit{var}}, \bar{x}) \\
\text{Expressions :} & \underline{\circ(e_1, e_2)} & = & \circ(\textbf{o}, \underline{e_1}, \underline{e_2}) \\
& \underline{hd(e_1)} & = & \circ(\textbf{\textit{hd}}, \underline{e_1}) \\
& \underline{tl(e_1)} & = & \circ(\textbf{\textit{tl}}, \underline{e_1}) \\
\text{Commandes :} & \underline{x \texttt{=} e_1} & = & \circ(\texttt{=}, \underline{x}, \underline{e_1}) \\
& \underline{c_1 \texttt{;} c_2} & = & \circ(\texttt{;}, \underline{c_1}, \underline{c_2})
\end{array}$$

Acceptabilité. Le langage Ker n'est pas acceptable au sens du chapitre 3. D'une part ce langage ne possède pas les capacités de calcul d'une machine de Turing : les programmes de Ker ne calculent que des fonctions calculables en temps constant. D'autre part, Ker n'admet pas d'auto-interpréteur. Pour montrer ce deuxième point il suffit d'observer que pour tout programme $\mathbf{q} \in \mathbb{P}^{\texttt{Ker}}$ et toute donnée $t \in \mathbb{D}$ il existe deux constantes $\alpha_1, \alpha_2 \in \mathbb{N}$ telles que

$$|\,[\![\mathbf{q}]\!]\,(t)\,| < \alpha_1 \cdot |t| + \alpha_2 \tag{7.1}$$

Cette équation se démontre par une récurrence directe sur les expressions et les commande de Ker.

On montre par l'absurde que Ker ne possède pas d'auto-interpréteur. Supposons que la fonction universelle de Ker soit calculable par un programme $\mathbf{uker} \in \mathbb{P}^{\texttt{Ker}}$. On prend le programme $\mathbf{q}$ défini par $work\,\texttt{=}\circ\,(work\,, work)\,;\,\mathbf{uker}$. Pour tout programme $\mathbf{p} \in \mathbb{P}^{\texttt{Ker}}$ il satisfait $[\![\mathbf{q}]\!]\,(\mathbf{p}) = [\![\mathbf{p}]\!]\,(\mathbf{p})$. D'après l'équation (7.1), il existe deux constantes $\alpha_1, \alpha_2 \in \mathbb{N}$ telles que

$$|\,[\![\mathbf{q}]\!]\,(t)\,| < \alpha_1 \cdot |t| + \alpha_2$$

On pose $\mathbf{p}$ le programme de Ker suivant.

$$work\,\texttt{=}\circ\,(\underbrace{work\,, \cdots, work}_{\alpha_1 \text{ fois}}, \underbrace{\bullet, \cdots, \bullet}_{\alpha_2 \text{ fois}})$$

On observe que $|\,[\![\mathbf{p}]\!]\,(\mathbf{p})\,| > \alpha_1 \cdot |\mathbf{p}| + \alpha_2$ et il suit

$$\begin{aligned}
&\alpha_1 \cdot |\mathbf{p}| + \alpha_2 > |\,[\![\mathbf{q}]\!]\,(\mathbf{p})\,| \\
&\alpha_1 \cdot |\mathbf{p}| + \alpha_2 > |\,[\![\mathbf{p}]\!]\,(\mathbf{p})\,| < \alpha_1 \cdot |\mathbf{p}| + \alpha_2
\end{aligned}$$

Ce qui est absurde. On conclut que la fonction universelle de Ker n'est pas calculable dans Ker.

Spécialisation. Par contre, Ker possède un spécialiseur. En effet, le spécialiseur **spec** de While est un programme de Ker. On rappelle que ce programme s'écrit de la manière suivante.

$$work\,\texttt{=}\circ\,(\texttt{;}, \circ(\texttt{=}, \underline{work}, \textbf{\textit{quote}}, tl\,(work)), hd\,(work))\,; \qquad (\textbf{spec})$$

De plus, pour tout programme $\mathbf{p} \in \mathbb{P}^{\texttt{Ker}}$ et pour toute donnée $t \in \mathbb{D}$ l'application du spécialiseur **spec** à l'entrée $\circ(\mathbf{p}, t)$ donne le programme

$$work\,\texttt{=}\circ\,(t, work)\,;\,\mathbf{p}$$

qui est bien un programme de Ker. Ainsi on a pour tout programme $\mathbf{p} \in \mathbb{P}^{\texttt{Ker}}$ et toutes données $t, u \in \mathbb{D}$

$$[\![\mathbf{p}_t]\!]\,(u) = [\![\mathbf{p}]\!]\,(t, u) \qquad \text{avec } \mathbf{p}_t = \textit{spec}(\mathbf{p}, t)$$

7.2 Mécanismes viraux dans Ker

Virus blueprint. Même si Ker n'est pas acceptable, il satisfait les théorème de récursion et de récursion explicite.

Théorème 16 (Théorème de récursion)**.** *Pour tout programme* $\mathbf{r} \in \mathbb{P}^{\mathit{Ker}}$ *il existe un programme* $\mathbf{e} \in \mathbb{P}^{\mathit{Ker}}$ *tel que pour toute donnée* $t \in \mathbb{D}$ *on a*

$$[\![\mathbf{e}]\!]\,(t) = [\![\mathbf{r}]\!]\,(\mathbf{e}, t)$$

Démonstration. On suit une démonstration que celle du théorème de récursion. On reprend le programme **k**.

$$\begin{array}{l} x = tl\,(work)\,; \\ work = \mathbf{spec}(hd\,(work))\,; \\ work = \circ\,(work\,, x)\,; \\ x = \bullet \end{array} \qquad (\mathbf{k})$$

On constate que **k** est bien un programme de Ker. Par conséquent, pour tout programme $\mathbf{r} \in \mathbb{P}^{\mathtt{Ker}}$, le programme $\mathbf{q} = \mathbf{k}\,;\mathbf{r}$ est un programme de Ker et par suite le point fixe $\mathbf{e} = \mathit{spec}(\mathbf{q}, \mathbf{q})$ est aussi un programme de Ker. On observe que pour toute donnée $t \in \mathbb{D}$, on a la dérivation suivante

$$\begin{array}{rll} [\![\mathbf{e}]\!]\,(t) = [\![\mathit{spec}(\mathbf{k}\,;\mathbf{r}, \mathbf{k}\,;\mathbf{r})]\!]\,(t) & & \text{par définition de } \mathbf{e} \\ = [\![\mathbf{k}\,;\mathbf{r}]\!]\,(\mathbf{k}\,;\mathbf{r}, t) & & \text{par définition d'un spécialiseur} \\ = [\![\mathbf{r}]\!]\,\big(\mathit{spec}(\mathbf{k}\,;\mathbf{r}, \mathbf{k}\,;\mathbf{r}), t\big) & & \text{par définition de } \mathbf{q} \\ = [\![\mathbf{r}]\!]\,(\mathbf{e}, t) & & \text{par définition de } \mathbf{e} \end{array}$$

On conclut que Ker satisfait le théorème de récursion. □

Théorème 17 (Théorème de récursion explicite)**.** *Pour tout programme* $\mathbf{r} \in \mathbb{P}^{\mathit{Ker}}$ *il existe un programme* $\mathbf{e} \in \mathbb{P}^{\mathit{Ker}}$ *tel que la fonction* $\mathcal{E} = [\![\mathbf{e}]\!]$ *satisfait*

$$[\![\mathcal{E}(u)]\!]\,(t) = [\![\mathbf{r}]\!]\,(\mathbf{e}, u, t) \qquad (7.2)$$

Démonstration. On reprend le programme **xpl**.

$$\begin{array}{l} x = \mathbf{spec}(\mathbf{spec}, hd\,(work))\,; \\ work = \circ\,(x, tl\,(work)) \end{array} \qquad (\mathbf{xpl})$$

On constate que **xpl** est bien un programme de Ker. Pour tout programme $\mathbf{r} \in \mathbb{P}^{\mathtt{Ker}}$, le programme $\mathbf{q} = \mathbf{xpl}\,;\mathbf{r}$ est aussi un programme de Ker. Pour toutes données $w, u, t \in \mathbb{D}$ on a

$$[\![\mathbf{q}]\!]\,(w, u, t) = [\![\mathbf{r}]\!]\,\big(\mathit{spec}(\mathbf{spec}, w), u, t\big)$$

Par application du théorème de récursion au programme $\mathbf{q}$, on obtient un programme $\mathbf{r}' \in \mathbb{P}^{\mathtt{Ker}}$ tel que pour toutes données $u, t \in \mathbb{D}$

$$[\![\mathbf{r}']\!]\,(u,t) = [\![\mathbf{q}]\!]\,(\mathbf{r}',u,t) = [\![\mathbf{r}]\!]\,\big(\mathit{spec}(\mathbf{spec},\mathbf{r}'),u,t\big)$$

On pose $\mathbf{e} = \mathit{spec}(\mathbf{spec},\mathbf{r}')$ et $\mathcal{E} = [\![\mathbf{e}]\!]$. On observe que $\mathbf{e}$ est bien un programme de Ker. Comme $\mathcal{E}(u) = \big[\![\mathit{spec}(\mathbf{spec},\mathbf{r}')]\!\big]\,(u) = \mathit{spec}(\mathbf{r}',u)$, Il suit que pour toutes données $u, t \in \mathbb{D}$ on a

$$\begin{aligned}
[\![\mathcal{E}(u)]\!]\,(t) &= \big[\![\mathit{spec}(\mathbf{r}',u)]\!\big]\,(t)\\
&= [\![\mathbf{r}']\!]\,(u,t)\\
&= [\![\mathbf{r}]\!]\,\big(\mathit{spec}(\mathbf{spec},\mathbf{r}'),u,t\big)\\
&= [\![\mathbf{r}]\!]\,(\mathbf{e},u,t)
\end{aligned}$$

On conclut que le programme $\mathbf{e}$ satisfait le théorème. □

Nous avons vu dans la preuve de la propriété 5, que la construction des virus blueprint est fondée sur le théorème de récursion. Comme ce théorème est valable dans Ker on conclut que cette propriété l'est aussi.

Propriété 18. *Pour tout programme $\mathbf{r} \in \mathbb{P}^{\mathtt{Ker}}$ il existe un virus blueprint de code viral $\mathbf{v} \in \mathbb{P}^{\mathtt{Ker}}$ relativement au comportement $[\![\mathbf{r}]\!]$. C'est à dire que pour toute donnée $t \in \mathbb{D}$ on a*

$$[\![\mathbf{v}]\!]\,(t) = [\![\mathbf{r}]\!]\,(\mathbf{v},t)$$

Démonstration. Application directe du théorème de récursion. □

Cette propriété nous permet de corroborer l'observation du chapitre 4, à savoir qu'un langage de programmation peut admettre des virus informatiques même s'il ne comporte pas d'auto-interpréteur.

Virus Smith. Le cas des virus Smith est plus problématique. Certes, on peut conclure à l'existence d'un compilateur de virus Smith dans Ker en observant que la preuve de la propriété 8 est encore valable. Mais la construction des virus Smith a peu de sens dans ce contexte.

On rappelle qu'un tel virus est composé d'un code viral $\mathbf{v}$ et d'une fonction de propagation $\mathcal{B}$ satisfaisant pour tout hôte $\mathbf{p} \in \mathbb{P}^{\mathtt{Ker}}$ et toute donnée $t \in \mathbb{D}$

$$[\![\mathcal{B}(\mathbf{v},\mathbf{p})]\!]\,(t) = [\![\mathbf{r}]\!]\,(\mathbf{b},\mathbf{v},\mathbf{p},t) \qquad \text{avec } [\![\mathbf{b}]\!] = \mathcal{B}$$

Pour propager l'infection, la spécification $\mathbf{r}$ utilise le programme $\mathbf{b}$ qui calcule la fonction de propagation $\mathcal{B}$. Pour infecter un nouvel hôte, il est nécessaire d'interpréter ce programme $\mathbf{b}$. Or dans le cadre du langage Ker nous ne disposons pas d'auto-interpréteur. Ainsi, cette construction semble caduc.

Reproduction par vecteur d'infection. Nous considérons une version affaiblie de la reproduction par vecteur d'infection.

Définition 19 (Virus Smith faible)**.** Soient $\mathcal{F}, \mathit{slt}_1, \cdots, \mathit{slt}_n$ des fonctions de $\mathbb{D}$ vers $\mathbb{D}$. Un programme $\mathbf{v} \in \mathbb{D}$ associé à une fonction $\mathcal{B} : \mathbb{D} \to \mathbb{D}$ est un *virus Smith faible* relativement au comportement $\mathcal{F}$ et aux *fonctions de sélection* $\mathit{slt}_1, \cdots, \mathit{slt}_n$ si d'une part $\mathbf{v}$ est un virus relativement à la fonction de propagation $\mathcal{B}$ et d'autre part si pour tout programme $\mathbf{p} \in \mathbb{D}$ et toute donnée $t \in \mathbb{D}$ on a

$$[\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!](t) = \mathcal{F}(\mathcal{B}(\mathbf{v}, t_1), \cdots, \mathcal{B}(\mathbf{v}, t_n), \mathbf{v}, \mathbf{p}, t) \qquad \text{avec} \quad \begin{array}{r} t_1 = \mathit{slt}_1(t) \\ \cdots \\ t_n = \mathit{slt}_n(t) \end{array} \tag{7.3}$$

Dans cette définition, les fonctions de sélection $\mathit{slt}_1, \cdots, \mathit{slt}_n$ recherchent à travers l'état du système des hôtes afin de propager l'infection. Le comportement $\mathcal{F}$ utilise le code viral $\mathbf{v}$, l'hôte original $\mathbf{p}$, les formes infectées $\mathcal{B}(\mathbf{v}, \mathit{slt}_1(t)), \cdots, \mathcal{B}(\mathbf{v}, \mathit{slt}_n(t))$ et l'état t pour propager l'infection. On observe qu'ici la présence d'un auto-interpréteur n'est pas nécessaire. En quelque sorte, le programme de la fonction de propagation a été réaligné dans le code viral.

Construction de virus Smith faibles. La construction des virus Smith faibles repose sur une forme affaiblie du théorème de récursion explicite dont la formulation est très proche de celle du théorème de récursion étendu présenté par Smullyan dans [Smu93].

Théorème 20. *Pour tous programmes* $\mathbf{r}, \mathbf{q}_1, \cdots, \mathbf{q}_n : \mathbb{D} \to \mathbb{D}$ *il existe une fonction calculable* $\mathcal{E} : \mathbb{D} \to \mathbb{D}$ *telle que pour toutes données* $t, u \in \mathbb{D}$ *on a*

$$[\![\mathcal{E}(u)]\!](t) = [\![\mathbf{r}]\!](\mathcal{E}(t_1), \cdots, \mathcal{E}(t_n), u, t) \qquad \textit{avec} \quad \begin{array}{r} t_1 = [\![\mathbf{q}_1]\!](t) \\ \cdots \\ t_n = [\![\mathbf{q}_n]\!](t) \end{array} \tag{7.4}$$

Démonstration. A un renommage près, on suppose que les programmes $\mathbf{q}_1, \cdots, \mathbf{q}_n$ utilisent des variables distinctes. On se munit des variables $x, y, z, \alpha_1, \cdots, \alpha_n \in \mathbb{X}$ n'apparaissant pas dans les programmes $\mathbf{r}, \mathbf{q}_1, \cdots, \mathbf{q}_n$. On pose **wkx** le programme suivant.

$$\begin{array}{l} [x, y, z] = \mathit{work}\ ; \\ \alpha_1 = \mathbf{q}_1(z)\ ; \\ \alpha_1 = \mathbf{spec}(x, \alpha_1)\ ; \\ \cdots \\ \alpha_n = \mathbf{q}_n(z)\ ; \\ \alpha_n = \mathbf{spec}(x, \alpha_n)\ ; \\ \mathit{work} = \circ\,(\alpha_1, \cdots, \alpha_n, y, z)\ ; \end{array} \tag{\textbf{wkx}}$$

On observe que le programme $\mathbf{r}' = \mathbf{wkx}\,;\mathbf{r}$ satisfait pour toutes données $w, u, t \in \mathbb{D}$

$$[\![\mathbf{r}']\!]\,(w,u,t) = [\![\mathbf{r}]\!]\,\big(\mathit{spec}(w,t_1),\cdots,\mathit{spec}(w,t_n),u,t\big) \qquad \text{avec} \quad \begin{array}{r} t_1 = [\![\mathbf{q}_1]\!]\,(t) \\ \cdots \\ t_n = [\![\mathbf{q}_n]\!]\,(t) \end{array}$$

On applique le théorème de récursion au programme $\mathbf{r}'$ et on obtient un programme $\mathbf{d} \in \mathbb{D}$ tel que pour toutes données $w, u \in \mathbb{D}$ on a $[\![\mathbf{d}]\!]\,(u,t) = [\![\mathbf{r}']\!]\,(\mathbf{d},u,t)$. On pose la fonction $\mathcal{E}(u) = \mathit{spec}(\mathbf{d},u)$ et on observe que pour toutes données $t, u \in \mathbb{D}$ on a

$$\begin{aligned} [\![\mathcal{E}(u)]\!]\,(t) &= [\![\mathit{spec}(\mathbf{d},u)]\!]\,(t) \\ &= [\![\mathbf{d}]\!]\,(u,t) \\ &= [\![\mathbf{r}']\!]\,(\mathbf{d},u,t) \\ &= [\![\mathbf{r}]\!]\,\big(\mathit{spec}(\mathbf{d},t_1),\cdots,\mathit{spec}(\mathbf{d},t_n),u,t\big) \qquad \text{avec} \quad \begin{array}{r} t_1 = [\![\mathbf{q}_1]\!]\,(t) \\ \cdots \\ t_n = [\![\mathbf{q}_n]\!]\,(t) \end{array} \\ &= [\![\mathbf{r}]\!]\,(\mathcal{E}(t_1),\cdots,\mathcal{E}(t_n),u,t) \end{aligned}$$

On conclut que $\mathcal{E}$ satisfait le théorème. □

Dans la démonstration précédente, on notera que la construction de $\mathcal{E}$ est uniforme en $\mathbf{r}$. Toutefois, le renommage des variables nécessaire à la construction du programme **wkx** depuis les programmes $\mathbf{q}_1,\cdots,\mathbf{q}_n$ outrepasse les capacités de calcul de `Ker`. Par conséquent, nous n'avons pas de compilateur de point fixe relatif au théorème précédant dans le cadre du langage `Ker`. Nous verrons que nous avons tout de même une notion de quasi-compilation.

7.3 Quasi-compilation

Quasi-compilateur. Un *quasi-compilateur*[1] de point fixe pour le théorème de récursion 20 est un programme $\mathbf{qcomp} \in \mathbb{D}$ tel que pour tout programme $\mathbf{r} \in \mathbb{P}^{\mathtt{Ker}}$ il existe une donnée $\mathbf{r}' \in \mathbb{D}$ telle que l'exécution du quasi-compilateur **qcomp** sur $\mathbf{r}'$ retourne un programme $\mathbf{e} = [\![\mathbf{qcomp}]\!]\,(\mathbf{r}')$ satisfaisant pour toutes données $t, u \in \mathbb{D}$

$$[\![\mathcal{E}(u)]\!]\,(t) = [\![\mathbf{r}]\!]\,(\mathcal{E}(t_1),\cdots,\mathcal{E}(t_n),u,t) \qquad \text{avec} \quad \begin{array}{r} t_1 = [\![\mathbf{q}_1]\!]\,(t) \\ \cdots \\ t_n = [\![\mathbf{q}_n]\!]\,(t) \end{array}$$

Par exemple le programme suivant, noté **qcomp**, est un tel quasi-compilateur.

$$\begin{array}{l} \mathit{work} = \circ\,(;,\mathbf{k},\mathit{work})\,; \\ \mathit{work} = \mathbf{spec}(\mathit{work},\mathit{work})\,; \\ \mathit{work} = \mathbf{spec}(\mathbf{spec},\mathit{work}) \end{array} \qquad (\mathbf{qcomp})$$

[1] La terminologie « quasi » est empruntée à Smullyan [Smu94].

Pour toute donnée $t \in \mathbb{D}$ on a $[\![\mathbf{qcomp}]\!](t) = \mathit{spec}(\mathbf{spec}, \mathit{spec}(\mathbf{k}\,;t,\mathbf{k}\,;t))$. Pour tout programme $\mathbf{r} \in \mathbb{P}^{\mathtt{Ker}}$ on prend $\mathbf{r}' = \mathbf{wkx}\,;\mathbf{r}$ où $\mathbf{wkx}$ est le programme de la démonstration du théorème 20 obtenu par renommage des variables des programmes $\mathbf{q}_1, \cdots, \mathbf{q}_n$.

$$[\![\mathbf{qcomp}]\!](\mathbf{r}') = \mathbf{e} \quad [\![\mathbf{e}]\!](u) = \mathit{spec}(\mathbf{d}, u) \quad \text{avec } \mathbf{d} = \mathit{spec}(\mathbf{k}\,;\mathbf{wkx}\,;\mathbf{r}', \mathbf{k}\,;\mathbf{wkx}\,;\mathbf{r}')$$

La fonction $[\![\mathbf{e}]\!]$ correspond bien à la fonction $\mathcal{E}$ exhibé dans la démonstration.

On peut voir la quasi-compilation comme une compilation partielle requérant une aide extérieur. Au lieu de donner directement la spécification à compiler, un intervenant extérieur aux capacités de calcul supérieures opère un pré-travail. Ensuite la compilation est terminée de manière automatique par le quasi-compilateur.

La notion de quasi-compilateur est très proche des quasi-diagonaliseurs décrits dans [Smu94]. Dans cet ouvrage, Smullyan étudie des constructions de point fixe très génériques permettant de montrer les théorèmes de récursion dans des systèmes aux propriétés très restreintes.

Compilation de virus Smith faibles. La compilation de virus Smith faible est une application directe du théorème de récursion 20.

Propriété 21. *Pour tous programmes $\mathbf{p}, \mathbf{slt}_1, \cdots, \mathbf{slt}_n \in \mathbb{D}$ il existe un virus Smith relativement au comportement $[\![\mathbf{p}]\!]$ et aux fonctions de sélection $[\![\mathbf{slt}_1]\!], \cdots, [\![\mathbf{slt}_n]\!]$. C'est à dire que pour tout programme $\mathbf{p} \in \mathbb{D}$ et toute donnée $t \in \mathbb{D}$ on a*

$$[\![\mathcal{B}(\mathbf{v},\mathbf{p})]\!](t) = \mathcal{F}(\mathcal{B}(\mathbf{v},t_1), \cdots, \mathcal{B}(\mathbf{v},t_n), \mathbf{v}, \mathbf{p}, t) \qquad \begin{array}{ll} \textit{avec} & t_1 = \mathit{slt}_1(t) \\ & \cdots \\ & t_n = \mathit{slt}_n(t) \end{array} \tag{7.5}$$

Démonstration. On note $\mathbf{wrap}_2$ le programme suivant où $\alpha_1, \cdots, \alpha_n, x, y \in \mathbb{X}$ sont des nouvelles variables.

$$\begin{array}{l} [\alpha_1, \cdots, \alpha_n, x, y] = work\,; \\ work = \circ\,(\alpha_1, \cdots, \alpha_n, hd\,(x), tl\,(x), tl\,(y)) \end{array} \tag{$\mathbf{wrap}_2$}$$

Ce programme permet de réorganiser les arguments de l'entrée. On observe que pour tous $w_1, \cdots, w_n, u, \mathbf{p}, t \in \mathbb{D}$ le programme $\mathbf{wrap}_2\,;\mathbf{r}$ satisfait

$$[\![\mathbf{wrap}_2\,;\mathbf{r}]\!](w_1, \cdots, w_n, \circ(u,\mathbf{p}), \circ(u,t)) = [\![\mathbf{r}]\!](w_1, \cdots, w_n, u, \mathbf{p}, t)$$

On se munit d'une variable $z \in \mathbb{X}$ n'apparaissant dans aucun des programmes $\mathbf{slt}_1, \cdots, \mathbf{slt}_n$. Pour tout $m \in \{1, \cdots, n\}$ on note $\mathbf{slt}'_m$ le programme suivant.

$$\begin{array}{l} z = hd\,(work)\,; \\ work = \mathbf{slt}_m(tl\,(work))\,; \\ work = \circ\,(z, work)\,; \end{array} \tag{$\mathbf{slt}'_m$}$$

On observe que pour tout $m \in \{1, \cdots, n\}$ on a $[\![\mathbf{slt}'_m]\!]\,(u,t) = \circ(u, [\![\mathbf{slt}_m]\!]\,(t))$

On applique le théorème 20 aux programmes $\mathbf{wrap}\,;\mathbf{r}, \mathbf{slt}'_1, \cdots, \mathbf{slt}'_n$ et on obtient une fonction $\mathcal{B}$ telle que pour tous $u, \mathbf{p}, t \in \mathbb{D}$ on a

$$\begin{aligned}[\![\mathcal{B}(u,\mathbf{p})]\!]\,(t) &= [\![\mathbf{wrap}\,;\mathbf{r}]\!]\,(\mathcal{B}(u,t_1), \cdots, \mathcal{B}(u,t_n), \circ(u,\mathbf{p}), \circ(u,t)) \quad \text{avec} \quad t_1 = \mathit{slt}_1(t)\\ &= [\![\mathbf{r}]\!]\,(\mathcal{B}(u,t_1), \cdots, \mathcal{B}(u,t_n), u, \mathbf{p}, t) \qquad\qquad\qquad\qquad\quad \cdots\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad t_n = \mathit{slt}_n(t)\end{aligned}$$

Il reste à construire le code viral. Avec **wkx** l'instance du programme défini dans la démonstration du théorème 20 pour les programmes $\mathbf{slt}'_1, \cdots, \mathbf{slt}'_n$ et $\mathbf{e}$ le point fixe explicite, on pose $\mathbf{q}$ comme étant le programme suivant où x, y, z sont des nouvelles variables.

$$\begin{aligned}&[x,y,z] = work\,;\\ &work = \circ\,(\mathbf{e}, \circ(x,y), \circ(x,z))\,;\\ &\mathbf{wkx}\,;\\ &\mathbf{wrap}_2\,;\\ &\mathbf{r}\end{aligned}$$

On observe que ce programme satisfait pour tous $u, \mathbf{p}, t \in \mathbb{D}$

$$\begin{aligned}[\![\mathbf{q}]\!]\,(u,\mathbf{p},t) = [\![\mathbf{r}]\!]\,(\mathcal{B}(u,t_1), \cdots, \mathcal{B}(u,t_n), u, \mathbf{p}, t) \qquad \text{avec} \quad t_1 &= \mathit{slt}_1(t)\\ \cdots\\ t_n &= \mathit{slt}_n(t)\end{aligned}$$

On conclut en appliquant le théorème de récursion au programme $\mathbf{q}$. □

Dans cette démonstration on observe que la construction de virus Smith faibles requière une quasi-compilation. Par conséquent, il ne semble pas toujours possible de créer des virus Smith faibles par les seules capacités de calcul de `Ker`. Une aide extérieure est nécessaire, elle consiste principalement à éviter les collisions de variables entre les programmes $\mathbf{r}, \mathbf{slt}_1, \cdots, \mathbf{slt}_n$.

7.4 Conclusions

Nous avons adressé l'interrogation de Cohen en montrant que les capacités de calcul d'un système ne semble pas influer sur l'existence de virus informatiques. En effet, le langage `Ker` satisfait la majorité des théorèmes de récursion alors que ses capacités de calcul sont extrêmement limitées. Il est ainsi possible de construire des virus blueprint ou Smith relativement à tout comportement pouvant être spécifié dans `Ker`. Les briques élémentaires de ces constructions étant un spécialiseur et la possibilité de composer des bloques d'instructions, il semble difficile de limiter les fonctionnalités au point d'éliminer ces opérations.

Par ailleurs, nous avons été intrigué par le fait que la construction de virus Smith est possible alors que ces virus n'ont pas de sens dans le contexte de `Ker`. Suite

à cette observation nous avons établi une version affaiblie de la reproduction par vecteur d'infection donnant lieu à une nouvelle classe de virus : les virus Smith faibles. Nous avons observé que pour tout comportement de virus Smith faible pouvant être spécifié dans `Ker`, il existe un virus satisfaisant ce comportement. Néanmoins, la construction d'un tel virus ne semble pas toujours réalisable par les seules capacités de calcul de `Ker`.

Ce dernier point est un pas en avant vers la conception de systèmes libre de toute construction virale. En effet, on peut soupçonner l'existence de systèmes qui soient incapables de construire des virus informatiques. Ceci est intéressant dans le sens où si initialement le système ne possède pas de virus et que l'on n'introduit pas non plus de virus par une intervention extérieure, son évolution n'engendrera pas de virus. L'existence de tels systèmes est encore hypothétique mais elle reste une piste dans le domaine de la protection antivirale.

8 Virologie et complexité de Kolmogorov

Dans le chapitre précédent nous avons constaté qu'il ne suffisait pas de limiter les fonctionnalités d'un langage de programmation pour se prémunir des infections informatiques. En effet, l'existence de programmes auto-reproducteurs semble plutôt provenir d'une relation entre la sémantique des programmes et leur syntaxe concrète. Il existe un outil de l'informatique théorique reliant ces deux aspects : la complexité de Kolmogorov. Cet outil permet de quantifier la difficulté nécessaire à la construction d'une donnée. De manière intuitive la complexité de Kolmogorov d'une donnée correspond à la taille de la plus petite description de cette donnée. On constate que cette notion fait intervenir la sémantique, au niveau de la description, et la syntaxe concrète, au niveau de la taille. Depuis cette observation, nous étudions des stratégies de défenses fondées sur cet outil.

La première stratégie sollicite un mécanisme de compression. En comprimant les programmes, on fait en sorte que leur taille approche leur complexité de Kolmogorov. Or un programme étant une description de lui-même, on réduit ainsi l'espace disponible aux virus pour s'introduire à l'intérieur d'un hôte. En suivant ce principe nous construisons un système où la complexité de Kolmogorov des virus parasites est bornée. Cette propriété limite les infections possibles.

Dans une deuxième partie nous considérons une protection par contrôle des flux d'information. Le principe repose sur une hiérarchisation des données à laquelle on adjoint un langage de programmation satisfaisant une propriété d'intégrité. De manière intuitive, cette propriété interdit les flux d'information depuis les niveaux inférieurs vers les niveaux supérieurs. De ce fait, si un certain niveau du système est infecté, cette infection est circonscrite aux niveaux inférieurs. Les autres niveaux restant inaccessibles et immunisés.

Dans un troisième temps, nous montrons comment instaurer une telle politique par un typage du langage `While`. Ce travail est une adaptation naïve et simplifiée des études [Mye99] et [MB05] respectivement réalisées dans le cadre d'un langage impératif et d'un langage ML.

8.1 Complexité de Kolmogorov

Soient deux données $t, u \in \mathbb{D}$. La complexité de Kolmogorov de la donnée u sachant la donnée t quantifie la difficulté nécessaire à la construction de u depuis t. On représente cette complexité par la taille du plus petit programme retournant u depuis l'entrée t. Plus formellement on donne la définition suivante.

Définition 22. La *complexité de Kolmogorov* est la fonction $\mathcal{K} : \mathbb{D} \to \mathbb{N}$ définie pour toutes données $t, u \in \mathbb{D}$ par

$$\mathcal{K}(u \mid t) = \min \{|\mathbf{p}| \mid [\![\mathbf{p}]\!](t) = u\}$$

On dit que $\mathcal{K}(u \mid t)$ est la complexité de Kolmogorov de u sachant t.

On observe que pour toutes données $t, u \in \mathbb{D}$ et tout programme $\mathbf{p} \in \mathbb{D}$ tel que $[\![\mathbf{p}]\!](t) = u$, on a par définition $\mathcal{K}(u \mid t) \leq |\mathbf{p}|$.

Remarque 23. Si l'on remplace le symbole $\circ$ par le chiffre 1 et le symbole $\bullet$ par le chiffre 0 et si l'on représente les arbres binaires sous la forme de termes, on observe alors que ces arbres constituent un code préfixe tout à fait usuel. Ainsi la complexité de Kolmogorov que nous avons définie correspond bien à une complexité de Kolmogorov en version préfixe.

Complexité de Kolmogorov intrinsèque. Par la suite nous aurons besoin d'une notion de complexité intrinsèque. On l'obtient en fixant la donnée relativement à laquelle on calcule la complexité de Kolmogorov.

Définition 24. La *complexité intrinsèque de Kolmogorov* est la fonction $\mathcal{K}_0 : \mathbb{D} \to \mathbb{N}$ définie par

$$\mathcal{K}_0(u) = \mathcal{K}_0(u \mid \bullet) = \min \{|\mathbf{p}| \mid [\![\mathbf{p}]\!](\bullet) = u\}$$

On relie la complexité intrinsèque à la complexité de Kolmogorov par les propriétés suivantes.

Propriété 25. *Il existe une constante $\alpha \in \mathbb{N}$ telle que pour toutes données $t, u \in \mathbb{D}$*

$$\mathcal{K}(u \mid t) \leq \mathcal{K}_0(u) + \alpha$$

Démonstration. Soient $t, u \in \mathbb{D}$ des données et $\mathbf{p} \in \mathbb{D}$ un programme tel que $[\![\mathbf{p}]\!](\bullet) = u$. Le programme $work = \bullet \,;\mathbf{p}$ satisfait $[\![work = \bullet \,;\mathbf{p}]\!](t) = u$. Par conséquent, on a $\mathcal{K}(u \mid t) \leq |\, work = \bullet \,;\mathbf{p}|$. Or la taille de la commande $work = \bullet$ est fixée, on en déduit la constante α. □

Propriété 26. *Il existe une constante $\alpha \in \mathbb{N}$ telle que pour toutes données $t, u \in \mathbb{D}$*

$$\mathcal{K}_0(u) \leq \mathcal{K}(u \mid t) + \mathcal{K}_0(t) + \alpha$$

Démonstration. Soient $t, u \in \mathbb{D}$ des données et $\mathbf{p}, \mathbf{q} \in \mathbb{D}$ des programmes tels que $[\![\mathbf{p}]\!](t) = u$, $[\![\mathbf{q}]\!](\bullet) = t$, $|\mathbf{p}| = \mathcal{K}(u \mid t)$ et $|\mathbf{q}| = \mathcal{K}_0(t)$. A un renommage des variables près, on peut supposer que $\mathbf{p}$ et $\mathbf{q}$ utilisent des variables distinctes. Le programme $\mathbf{q}\,;\mathbf{p}$ satisfait $[\![\mathbf{q}\,;\mathbf{p}]\!](\bullet) = u$. Par conséquent, on a $\mathcal{K}_0(u) \leq |\mathbf{q}\,;\mathbf{p}|$. Or la taille de ce programme satisfait $|\mathbf{q}\,;\mathbf{p}| = |\mathbf{p}| + |\mathbf{q}| + \alpha$ pour une certaine constante $\alpha \in \mathbb{N}$. On conclut que $\mathcal{K}_0(u) \leq \mathcal{K}(u \mid t) + \mathcal{K}(t) + \alpha$. □

8.2 Protection contre les virus parasite

Virus parasite. Dans les chapitres précédents, nous avons vu qu'un virus parasite préserve son hôte au cours du processus d'infection. Nous formalisons cette notion dans le cadre de la complexité de Kolmogorov en disant qu'un programme infecté par un virus parasite contient d'une part la description du code viral et d'autre part la description de l'hôte. Ainsi, pour tout virus parasite composé d'un code viral $\mathbf{v} \in \mathbb{D}$ et d'une fonction de propagation $\mathcal{B} : \mathbb{D} \to \mathbb{D}$ il existe une constante $\gamma \in \mathbb{N}$ telle que

$$\mathcal{K}_0(\mathbf{v}) + \mathcal{K}_0(\mathbf{p}) \leq |\mathcal{B}(\mathbf{v}, \mathbf{p})| + \gamma \tag{8.1}$$

Compression de programmes. Pour compresser un programme $\mathbf{p} \in \mathbb{D}$, il suffit de prendre une de ses meilleurs descriptions, c'est à dire un programme $\mathbf{q} \in \mathbb{D}$ satisfaisant $|\mathbf{q}| = \mathcal{K}_0(\mathbf{p})$ et $[\![\mathbf{q}]\!](\bullet) = \mathbf{p}$. Ensuite, on remplace $\mathbf{p}$ par un programme qui exécute $\mathbf{q}$ pour reconstruire $\mathbf{p}$ puis exécute $\mathbf{p}$ sur l'entrée originale. La propriété suivante formalise cette méthode de compression.

Propriété 27. *Il existe deux constantes $\alpha, \beta \in \mathbb{N}$ telles que pour tout programme $\mathbf{p} \in \mathbb{D}$ il y a un programme $\mathbf{p}' \in \mathbb{D}$ satisfaisant*

$$[\![\mathbf{p}']\!](\bullet) = [\![\mathbf{p}]\!] \qquad |\mathbf{p}'| \leq \mathcal{K}_0(\mathbf{p}) + \alpha \qquad \mathcal{K}(\mathbf{p}' \mid \mathbf{p}) = \beta \tag{8.2}$$

Démonstration. Soient $x, y \in \mathbb{X}$ deux variables n'apparaissant pas dans l'auto-interpréteur **univ**, on pose $\mathbf{r}$ le programme suivant.

$$\begin{array}{l} [x, y] = work\,; \\ work = \mathbf{univ}(x, \bullet)\,; \\ work = \mathbf{univ}(work\,, y) \end{array} \tag{$\mathbf{r}$}$$

On observe que pour tout programme $\mathbf{p} \in \mathbb{D}$ et toute donnée $t \in \mathbb{D}$ le programme $\mathbf{r}$ satisfait $[\![\mathbf{r}]\!](\mathbf{p}, t) = [\![[\![\mathbf{p}]\!](\bullet)]\!](t)$.

Soit $\mathbf{q} \in \mathbb{D}$ un programme tel que $|\mathbf{q}| = \mathcal{K}_0(\mathbf{p})$ et $[\![\mathbf{q}]\!](\bullet) = \mathbf{p}$. On pose $\mathbf{p}' = \mathit{spec}(\mathbf{r}, \mathbf{q})$. Par les définitions de $\mathbf{r}$ et $\mathbf{q}$, on a $[\![\mathbf{p}']\!] = [\![\mathbf{p}]\!]$. De plus le programme $\mathbf{p}'$ s'écrit $work = \circ\,(\mathbf{q}, work)\,;\mathbf{r}$. Comme le programme $\mathbf{r}$ est fixe, quel que soit le programme $\mathbf{p} \in \mathbb{D}$ on a $|\mathbf{p}'| \leq |\mathbf{q}| + \alpha$ où $\alpha \in \mathbb{N}$ est une constante. Il suit que $|\mathbf{p}'| \leq \mathcal{K}_0(\mathbf{p}) + \alpha$.

Avec la syntaxe concrète de `While` donnée au chapitre 2, le programme $\mathbf{r}'$ suivant satisfait $[\![\mathbf{r}']\!]\,(\mathbf{p}') = \mathbf{p}$.

$$work = tl\,(hd\,(tl^3\,(hd\,(tl\,(work))))) \qquad (\mathbf{r}')$$

On conclut que $\mathcal{K}(\mathbf{p} \mid \mathbf{p}') \leq |\mathbf{r}'|$ où $\mathbf{r}'$ est un programme fixé. On en déduit la constante β. □

Pour tout couple d'entiers $(\alpha, \beta) \in \mathbb{N}^2$, un système est (α, β)-*compressé* si pour tout programme $\mathbf{p}' \in \mathbb{D}$ présent sur le système, il existe un programme $\mathbf{p} \in \mathbb{D}$ satisfaisant l'équation (8.2).

Protection par compression. On se munit d'un couple d'entiers $(\alpha, \beta) \in \mathbb{N}^2$ et on considère la stratégie de protection suivante. Le système est (α, β)-compressé et on interdit que la taille des programmes croisse. C'est-à-dire que si un programme $\mathbf{p} \in \mathbb{D}$ du système est remplacé par un programme $\mathbf{q} \in \mathbb{D}$ alors $|\mathbf{q}| \leq |\mathbf{p}|$.

Sous ces conditions, nous montrons que la complexité de Kolmogorov du code viral $\mathbf{v} \in \mathbb{D}$ d'un virus parasite est bornée. Par conséquent, on limite de manière significative la complexité des infections possibles.

Supposons qu'un virus parasite infecte un hôte $\mathbf{p}' \in \mathbb{D}$. Par la propriété de compression du système, il y a un programme $\mathbf{p} \in \mathbb{D}$ tel que $|\mathbf{p}'| \leq \mathcal{K}(\mathbf{p}) + \alpha$ et $\mathcal{K}(\mathbf{p}' \mid \mathbf{p}) = \beta$. De plus, la contrainte de décroissance donne $|\mathcal{B}(\mathbf{v}, \mathbf{p}')| \leq |\mathbf{p}'|$. En effet le programme $\mathbf{p}'$ est remplacé par sa version infectée, la contrainte de décroissance impose que la taille de la forme infectée soit plus petite que celle de l'hôte original. En combinant ces équations, on obtient la dérivation suivante.

$$\begin{aligned} |\mathcal{B}(\mathbf{v}, \mathbf{p}')| &\leq |\mathbf{p}'| \\ |\mathcal{B}(\mathbf{v}, \mathbf{p}')| &\leq \mathcal{K}_0(\mathbf{p}) + \alpha \\ |\mathcal{B}(\mathbf{v}, \mathbf{p}')| - \mathcal{K}_0(\mathbf{p}') &\leq \mathcal{K}_0(\mathbf{p}) - \mathcal{K}_0(\mathbf{p}') + \alpha \end{aligned}$$

D'après la propriété 26, on a $\mathcal{K}_0(\mathbf{p}) - \mathcal{K}_0(\mathbf{p}') \leq \mathcal{K}(\mathbf{p} \mid \mathbf{p}')$. Comme $\mathcal{K}(\mathbf{p} \mid \mathbf{p}') = \beta$, il suit

$$|\mathcal{B}(\mathbf{v}, \mathbf{p}')| - \mathcal{K}_0(\mathbf{p}') \leq \alpha + \beta$$

En incluant l'équation (8.1) de la définition d'un virus parasite on obtient

$$\mathcal{K}_0(\mathbf{v}) \leq \alpha + \beta + \gamma$$

On observe que la complexité de Kolmogorov du code viral $\mathbf{v}$ est bornée par $\alpha+\beta+\gamma$.

Remarques. Pour utiliser cette stratégie de défense en pratique, il est nécessaire de pouvoir compresser les programmes de manière automatique. Or, il n'existe pas de procédure compression effectivement calculable satisfaisant les contraintes de

la propriété 27. Pour plus de détail sur ce point, le lecteur pourra consulter l'ouvrage [LV97] et plus particulièrement les chapitres traitant de l'incompressibilité.

Par conséquent, il semble difficile d'appliquer cette stratégie. Néanmoins, nous pouvons travailler avec une approximation de la procédure de compression sur un ensemble borné de programmes. Avec une constante $\alpha \in \mathbb{N}$, pour tout programme $\mathbf{p} \in \mathbb{D}$ tel que $|\mathbf{p}| \leq \alpha$, on a de manière triviale $|\mathbf{p}| \leq \mathcal{K}(\mathbf{p}) + \alpha$. On comprend que sur un tel ensemble de programmes toute procédure de compression peut être utilisée pour appliquer la stratégie de protection que nous proposons. Le principe est alors que meilleur est la procédure de compression, moins il y a d'espace disponible aux infections parasites.

8.3 Protection par contrôle des flux d'information

Dans cette partie, nous proposons une méthode de protection fondée sur le contrôle des flux d'information. Nous verrons dans quelle mesure une telle politique protège le système.

Un système stratifié. Nous nous plaçons dans un scénario où les données résident sur des niveaux d'intégrité. L'état du système est composé de ces niveaux indexées par les éléments du domaine de calcul $\mathbb{D}$. Ainsi l'état est représentée par une application $\mathcal{S} : \mathbb{D} \to \mathbb{D}$ qui pour tout niveau $i \in \mathbb{D}$ donne la donnée $\mathcal{S}(i)$ représentant l'ensemble des paramètres de niveau i. Par la suite on nomme simplement *état* l'application de $\mathbb{D} \to \mathbb{D}$ décrivant l'état du système.

Dans ce contexte, les programmes ne calculent plus sur des données mais sur un ensemble stratifié de données. En effet, un programme prend en entrée un état de $\mathbb{D} \to \mathbb{D}$ et son exécution retourne un nouvelle état de $\mathbb{D} \to \mathbb{D}$. Il paraît raisonnable de se limiter aux états finis, c'est-à-dire que seul un nombre fini de niveaux contiennent effectivement des données. Plus formellement, un état $\mathcal{S} : \mathbb{D} \to \mathbb{D}$ est fini si l'ensemble $\{i \mid \mathcal{S}(i) \neq \bullet\}$ est fini. Grâce à cette contrainte, il est possible de représenter l'état du système par un élément de $\mathbb{D}$. Par exemple on peut prendre le codage suivant.

$$\circ(\circ(\mathcal{S}(t_1), t_1), \cdots, \circ(\mathcal{S}(t_n), t_n), \bullet)$$

où $t \in \{t_1, \cdots, t_n\}$ est l'ensemble des niveaux sur lesquels l'application $\mathcal{S}$ est différente de $\bullet$. On retrouve ainsi la notion de langage de programmation définie au chapitre 3. Par convenance, nous continuons à travailler sur des applications de $\mathbb{D} \to \mathbb{D}$ tout en gardant à l'esprit que la notion de langage de programmation peut être retrouvée par l'intermédiaire d'un codage.

Politique d'intégrité. On se munit d'une relation d'ordre $\prec$ sur $\mathbb{D}$ de clôture réflexive $\preccurlyeq$ définissant un treillis sur $\mathbb{D}$ dont les opérations d'union et d'intersec-

tion sont notées $\curlyvee$ et $\curlywedge$. On rappelle qu'une telle structure satisfait les équations suivantes pour toutes données $t, u, w \in \mathbb{D}$.

$$t \curlywedge u \preccurlyeq t \qquad t \curlywedge u \preccurlyeq u \qquad \big(w \preccurlyeq t \text{ et } w \preccurlyeq u\big) \Rightarrow w \preccurlyeq t \curlywedge u$$
$$t \preccurlyeq t \curlyvee u \qquad u \preccurlyeq t \curlyvee u \qquad \big(t \preccurlyeq w \text{ et } u \preccurlyeq w\big) \Rightarrow t \curlyvee u \preccurlyeq w$$

Ce treillis nous permet de comparer les niveaux de données.

Pour tout niveau $j \in \mathbb{D}$ et tout état $\mathcal{S} : \mathbb{D} \to \mathbb{D}$ on note $\mathcal{S}\rceil^j$ l'état résultant de la réinitialisation des niveaux inférieurs à j.

$$\mathcal{S}\rceil^j(i) = \begin{cases} \bullet & \text{si } i \preccurlyeq j \\ \mathcal{S}(j) & \text{sinon} \end{cases}$$

Un langage de programmation satisfait la politique d'intégrité si pour calculer une donnée d'un certain niveau, on ne peut pas extraire d'information d'une donnée de niveau inférieur. Plus formellement, on donne la définition suivante.

Définition 28 (Politique d'intégrité)**.** La *politique d'intégrité* est satisfaite si pour tout niveau $i \in \mathbb{D}$ et tous états $\mathcal{S}, \mathcal{T} : \mathbb{D} \to \mathbb{D}$ on a

$$\mathcal{K}(\mathcal{T}\rceil^i \mid \mathcal{S}) = \mathcal{K}(\mathcal{T}\rceil^i \mid \mathcal{S}\rceil^i)$$

Protection contre les virus. La stratification des données peut être vue comme un typage, par exemple une donnée $t \in \mathbb{D}$ résidant sur un niveaux $i \in \mathbb{D}$ peut être vue comme une donnée typée notée $t \diamond i$. On note $\mathbb{D} \diamond \mathbb{D}$ l'ensemble de ces données typées.

Pour toute donnée $t \in \mathbb{D}$ et tout niveau $i \in \mathbb{D}$ on note $[i \mapsto t]$ l'état défini par

$$[i \mapsto t](j) = \begin{cases} t & \text{si } i = j \\ \bullet & \text{sinon} \end{cases}$$

On peut considérer qu'un système est en partie protégé contre les virus si sa conception assure qu'un virus ne puisse se propager que vers des niveaux inférieurs au sien. Dans notre formalisme cette propriété se traduit de la manière suivante. Un code viral $\mathbf{v} \in \mathbb{D} \diamond \mathbb{D}$ est *circonscrit* si pour tous niveaux $i, j \in \mathbb{D}$ tels que $i \preccurlyeq j$ on a

$$\mathcal{K}([j \mapsto \mathbf{v}] \mid [\![\mathbf{v} \diamond i]\!](\mathcal{S})) = \mathcal{K}([j \mapsto \mathbf{v}] \mid \mathcal{S}\rceil^j) \tag{8.3}$$

Littéralement, la complexité de Kolmogorov du virus au niveau j sachant une infection au niveau i et égale à sa complexité ne sachant aucune information provenant des niveaux inférieurs à j. Intuitivement, il est aussi difficile d'introduire un virus au niveau j depuis un état infecté au niveau i que de l'introduire depuis l'état désinfecté résultant le ré-initialisation de tous les niveaux inférieurs à j. En d'autres termes,

un virus est incapable d'extraire des informations depuis les niveaux inférieurs afin de se reproduire vers un niveau supérieur. C'est en ce sens que l'on peut considérer que l'infection est circonscrite.

Dans le cadre d'une politique d'intégrité, on peut facilement concevoir un système satisfaisant cette propriété de circonscription. Il suffit de s'assurer que pour tout programme $\mathbf{p} \in \mathbb{D}$, tous niveaux $i, j \in \mathbb{D}$ tels que $j \prec i$ et tout état $\mathcal{S} : \mathbb{D} \to \mathbb{D}$ on a

$$[\![\mathbf{p} \diamond i]\!] (\mathcal{S})\rceil^j = \mathcal{S}\rceil^j \tag{8.4}$$

C'est-à-dire qu'un programme de niveau $i \in \mathbb{D}$ ne peut modifier que des données résidant sur des niveaux inférieurs. En effet, par la politique d'intégrité on a

$$\mathcal{K}([j \mapsto \mathbf{v}] \mid [\![\mathbf{v} \diamond i]\!] (\mathcal{S})) = \mathcal{K}([j \mapsto \mathbf{v}] \mid [\![\mathbf{v} \diamond i]\!] (\mathcal{S})\rceil^j)$$

Et par l'équation (8.4), il suit

$$\mathcal{K}([j \mapsto \mathbf{v}] \mid [\![\mathbf{v} \diamond i]\!] (\mathcal{S})\rceil^j) = \mathcal{K}([j \mapsto \mathbf{v}] \mid \mathcal{S}\rceil^j)$$

Ce qui conduit à l'équation (8.3).

On remarque que l'équation (8.4) ne suffit pas à garantir la circonscription. En effet, on pourrait envisager qu'un programme d'un niveau supérieur à j propage le code viral depuis le niveau i vers le niveau j. On aurait une propagation par effet de bord et la circonscription serait mise en défaut. La politique d'intégrité interdit ce genre de propagation : c'est ici que l'emploi de la complexité de Kolmogorov prend tout son sens. Dans la section suivante, nous montrons que l'on peut facilement modifier le langage `While` de façon à assurer ces propriétés.

Remarque de Cohen. Dans [Coh89], Cohen met en doute l'efficacité d'une telle politique quant à la circonscription des infections virales. Pour lui, le point faible est que l'infection peut toujours se propager vers les niveaux inférieurs. Par conséquent, si le niveau administrateur est corrompu alors tous les niveaux inférieurs le sont aussi. Cohen admet tout de même qu'une telle politique permet de limiter les risques d'infection, mais il souligne que cela n'élimine pas pour autant tout risque.

8.4 Typage de `While`.

Dans cette partie, nous montrons comment instaurer la politique de sécurité présentée précédemment. Pour cela nous définissons une version typée du langage `While` nommé `TWhile`.

Syntaxe. On se muni d'un ensemble de variables typées $\mathbb{X} \diamond \mathbb{D}$ composés de couples $x \diamond i$ où $x \in \mathbb{X}$ est une variable et $i \in \mathbb{D}$ représente un niveau. Nous redéfinissons

la notion d'assignation comme une application de $\mathbb{X} \diamond \mathbb{D}$ vers $\mathbb{D} \diamond \mathbb{D}$. On note $\mathbb{S} \diamond \mathbb{D}$ l'ensemble des assignations typées.

La syntaxe de `TWhile` est tout à fait analogue à celle de `While` à la seule différence qu'on inclut le type des données, des variables et des programmes.

$$\begin{array}{rrcl}
\text{Domaine :} & \mathbb{D} \diamond \mathbb{D} & ::= & \{t \diamond i \mid t, i \in \mathbb{D}\} \\
\text{Variables :} & \mathbb{X} \diamond \mathbb{D} & ::= & \{x \diamond i \mid x \in \mathbb{X}, i \in \mathcal{S}\} \\
\text{Expressions :} & \mathbb{E} & ::= & \mathbb{D} \diamond \mathbb{D} \mid \mathbb{X} \diamond \mathbb{D} \mid \circ(\mathbb{E}, \mathbb{E}) \mid hd(\mathbb{E}) \mid tl(\mathbb{E}) \\
\text{Commandes :} & \mathbb{C} & ::= & \mathbb{X} \texttt{=} \mathbb{E} \mid \mathbb{C} \,;\, \mathbb{C} \mid while(\mathbb{E})\{\mathbb{C}\} \\
\text{Programmes :} & \mathbb{P} \diamond \mathbb{D} & ::= & \mathbb{C} \diamond \mathbb{D}
\end{array}$$

On y associe la syntaxe concrète suivante.

$$\begin{array}{rrcl}
\text{Domaine :} & \underline{t \diamond i} & = & \circ(\textbf{\textit{quote}}, t, i) \\
\text{Variables :} & \underline{x \diamond i} & = & \circ(\textbf{\textit{var}}, \bar{x}, i) \\
\text{Expressions :} & \underline{\circ(e_1, e_2)} & = & \circ(\texttt{o}, \underline{e_1}, \underline{e_2}) \\
 & \underline{hd(e_1)} & = & \circ(\textbf{\textit{hd}}, \underline{e_1}) \\
 & \underline{tl(e_1)} & = & \circ(\textbf{\textit{tl}}, \underline{e_1}) \\
 & \underline{e_1 \texttt{==} e_2} & = & \circ(\texttt{==}, \underline{e_1}, \underline{e_2}) \\
\text{Commandes :} & \underline{x \texttt{=} e_1} & = & \circ(\texttt{=}, \underline{x}, \underline{e_1}) \\
 & \underline{c_1 \,;\, c_2} & = & \circ(\texttt{;}, \underline{c_1}, \underline{c_2}) \\
 & \underline{while(e)\{c\}} & = & \circ(\textbf{\textit{while}}, \underline{e}, \underline{c}) \\
\text{Programmes :} & \underline{\mathbf{p} \diamond i} & = & \circ(\underline{\mathbf{p}}, i)
\end{array}$$

Comme précédemment, on ne fera pas de distinction entre un programme et sa syntaxe concrète.

Contrainte de typage. Les types des expressions et des commandes de `TWhile` sont récursivement définies par les équations suivantes.

$$\tau(e) = \begin{cases} i & \text{si } e = t \diamond i \text{ ou } e = x \diamond i \\ \curlywedge\{\tau(e_1), \tau(e_2)\} & \text{si } e = \circ(e_1, e_2) \text{ ou } e = e_1 \texttt{==} e_2 \end{cases}$$

$$\tau(c) = \begin{cases} \tau(e) & \text{si } c = x \texttt{=} e \\ \curlyvee\{\tau(c_1), \tau(c_2)\} & \text{si } c = c_1 \,;\, c_2 \\ \tau(e) & \text{si } c = while(e)\{c'\} \end{cases}$$

On définit une contrainte de typage afin de garantir la politique d'intégrité. On impose que le type du corps d'une boucle *while* soit inférieur au type de l'expression de contrôle. Plus formellement on définit récursivement les commandes bien typées comme suit.

– Pour toute variables $x \in \mathbb{X}$ et toute expression $e \in \mathbb{E}$, la commande $x \texttt{=} e$ est bien typée.

– Pour toutes commandes bien typées $c_1, c_2 \in \mathbb{C}$, la commande $c_1 \,;\, c_2$ est bien typée.
– Pour toute expression $e \in \mathbb{E}$ et toute commande bien typée $c \in \mathbb{C}$, si $\tau(c) \prec \tau(e)$ alors $\mathit{while}\,(e)\{c\}$ est bien typée.

Un programme $\mathbf{p} \diamond i \in \mathbb{D} \diamond \mathbb{D}$ est bien typé si $\mathbf{p}$ est bien typé et si $\tau(\mathbf{p}) \preccurlyeq i$. Dorénavant nous ne considérons que des programmes bien typés.

Sémantique Nous définissons la sémantique de `TWhile` en nous reposant sur les contraintes de typage.

$$
\begin{array}{rrcl}
\text{Domaine :} & (\!|t \diamond i|\!)\,(\sigma) & = & t \diamond i \\
\text{Variables :} & (\!|x|\!)\,(\sigma) & = & \sigma(x) \\
\text{Expressions :} & (\!|\circ(e_1,e_2)|\!)\,(\sigma) & = & \circ(t_1,t_2) \diamond i \text{ avec } \begin{array}{l} t_1 \diamond i_1 = (\!|e_1|\!)\,(\sigma) \\ t_2 \diamond i_2 = (\!|e_2|\!)\,(\sigma) \\ i = \tau(\circ(e_1,e_2)) \end{array} \\
& (\!|hd\,(e)|\!)\,(\sigma) & = & \begin{cases} \bullet \diamond i & \text{si } (\!|e|\!)\,(\sigma) = \bullet \diamond i \\ t_1 \diamond i & \text{si } (\!|e|\!)\,(\sigma) = \circ(t_1,t_2) \diamond i \end{cases} \\
& (\!|tl\,(e)|\!)\,(\sigma) & = & \begin{cases} \bullet \diamond i & \text{si } (\!|e|\!)\,(\sigma) = \bullet \diamond i \\ t_2 \diamond i & \text{si } (\!|e|\!)\,(\sigma) = \circ(t_1,t_2) \diamond i \end{cases} \\
& (\!|e_1 == e_2|\!)\,(\sigma) & = & \begin{cases} \circ(\bullet,\bullet) \diamond i & \text{si } (\!|e_1|\!)\,(\sigma) = (\!|e_2|\!)\,(\sigma) \text{ avec } i = \tau(e_1 == e_2) \\ \bullet \diamond i & \text{sinon avec } i = \tau(e_1 == e_2) \end{cases} \\
\text{Commandes :} & [\![x = e]\!]\,(\sigma) & = & \sigma[x \diamond i \mapsto t \diamond i] \text{ avec } t \diamond i = (\!|e|\!)\,(\sigma) \\
& [\![c_1 \,;\, c_2]\!]\,(\sigma) & = & [\![c_2]\!]\,([\![c_1]\!]\,(\sigma)) \\
& [\![\mathit{while}\,(e)\{c\}]\!]\,(\sigma) & = & \begin{cases} \sigma & \text{si } (\!|e|\!)\,(\sigma) = \bullet \diamond i \\ [\![\mathit{while}\,(e)\{c\}]\!]\,([\![c]\!]\,(\sigma)) & \text{sinon} \end{cases}
\end{array}
$$

Afin de définir la sémantique d'un programme nous utilisons les notations suivantes.

– Pour tout état $\mathcal{S} : \mathbb{D} \to \mathbb{D}$, on note $\mathit{In}(\mathcal{S})$ l'assignation $\sigma \in \mathbb{S} \diamond \mathbb{D}$ définie pour tout niveau $i \in \mathbb{D}$ par $\sigma(\mathit{work} \diamond i) = \mathcal{S}(i) \diamond i$ pour toute variable $x \in \mathbb{X}$ différente de work on a $\sigma(x \diamond i) = \bullet$.
– Pour toute assignation $\sigma \in \mathbb{S} \diamond \mathbb{D}$, on note $\mathit{Out}(\sigma)$ l'état $\mathcal{S} : \mathbb{D} \to \mathbb{D}$ défini pour tout niveau $i \in \mathbb{D}$ par $\mathcal{S}(i) = t$ où $(\!|\mathit{work} \diamond i|\!)\,(\sigma) = t \diamond i$.

De manière intuitive, $\mathit{In}(\mathcal{S})$ est l'assignation résultant de l'association de toute variable $\mathit{work} \diamond i \in \mathbb{X} \diamond \mathbb{D}$ à la donnée de niveau i dans l'état $\mathcal{S}$. De manière réciproque, $\mathit{Out}(\sigma)$ désigne l'état qui à tout niveau $i \in \mathbb{D}$ associe la valeur de la variable $\mathit{work} \diamond i$.

La sémantique d'un programme dépend de son typage : s'il n'est pas bien typé alors le programme est considéré comme non valide et il ne fait rien. Sinon, on l'exécute normalement.

$$
\text{Programmes :} \qquad [\![\mathbf{p} \diamond i]\!]\,(\mathcal{S}) = \begin{cases} \mathcal{S} & \text{si } \mathbf{p} \diamond i \text{ n'est pas bien typé} \\ \mathit{Out}(\sigma) & \text{sinon avec } \sigma = [\![\mathbf{p}]\!]\,(\mathit{In}(\mathcal{S})) \end{cases}
$$

Intégrité. Nous montrons que le langage `TWhile` satisfait la propriété d'intégrité.

Propriété 29. *Pour toute expression $e \in \mathbb{E}$ et toute assignation $\sigma \in \mathbb{S} \diamond \mathbb{D}$ on a pour tout niveau $i \prec \tau(e)$*

$$(\!|e|\!)(\sigma) = (\!|e|\!)(\sigma\rceil^i)$$

Démonstration. Nous montrons cette propriété par récurrence sur la forme d'une expression $e \in \mathbb{E}$ de type $j \in \mathbb{D}$ tel que $i \prec j$.

Cas $t \diamond j \in \mathbb{D} \diamond \mathbb{D}$. Immédiat puisque $(\!|e|\!)(\sigma)$ ne dépend pas de σ.

Cas $x \diamond j \in \mathbb{X} \diamond \mathbb{D}$. On $(\!|e|\!)(\sigma) = \sigma(x \diamond j)$ or par définition $\sigma\rceil^i(x \diamond j) = \sigma(x \diamond j)$.

Cas $hd(e_1)$ *ou* $tl(e_1)$. On rappelle que $\tau(e) = \tau(e_1)$. Par hypothèse de récurrence $(\!|e_1|\!)(\sigma) = (\!|e_1|\!)(\sigma\rceil^i)$.

Cas $\circ(e_1, e_2)$ *ou* $e_1 == e_2$. On rappelle que $\tau(e) = \curlywedge\{\tau(e_1), \tau(e_2)\}$. On en déduit que, comme $i \prec \tau(e)$, on a $i \prec \tau(e_1)$ et $i \prec \tau(e_2)$. Par hypothèse de récurrence on a $(\!|e_1|\!)(\sigma) = (\!|e_1|\!)(\sigma\rceil^i)$ et $(\!|e_2|\!)(\sigma) = (\!|e_2|\!)(\sigma\rceil^i)$. On conclut que $(\!|e|\!)(\sigma) = (\!|e|\!)(\sigma\rceil^i)$. □

Propriété 30. *Pour toute commande $c \in \mathbb{C}$ et tout niveau $i \in \mathbb{D}$ tels que $\tau(c) \preccurlyeq i$ il existe une commande $c' \in \mathbb{C}$ telle que $\tau(c') = \tau(c)$, $|c'| \leq |c|$ et pour tout magasin $\sigma \in \mathbb{S} \diamond \mathbb{D}$ on a*

$$[\![c']\!](\sigma\rceil^i) = \sigma'\rceil^i \text{ avec } \sigma' = [\![c]\!](\sigma)$$

Démonstration. Nous montrons cette propriété par récurrence sur la forme de c.

Cas $x = e$. On rappelle que $\tau(x = e) = \tau(e)$. Par conséquent $\tau(e) \preccurlyeq i$ et la commande $x = \bullet \diamond \tau(e)$ satisfait la propriété. Sinon, d'après la propriété 29 on a $(\!|e|\!)(\sigma\rceil^i) = (\!|e|\!)(\sigma)$, il suffit de prendre $c' = c$ pour satisfaire la propriété.

Cas $c_1 ; c_2$. Immédiat par hypothèse de récurrence.

Cas $while(e)\{c_1\}$. On rappelle que par contrainte de typage on a $\tau(c) \prec \tau(e)$ et de plus $\tau(while(e)\{c\}) = \curlyvee\{\tau(e), \tau(c)\}$. Ce qui mène à $\tau(while(e)\{c\}) = \tau(e)$

- Si $\tau(e) \preccurlyeq i$ alors seules les variables de type inférieur à i sont modifiées. Par conséquent, la commande $work = \bullet \diamond \tau(e)$ satisfait la propriété.
- Sinon, d'après la propriété 29 on a $(\!|e|\!)(\sigma\rceil^i) = (\!|e|\!)(\sigma)$ et par hypothèse de récurrence, il existe une commande c_1' telle que $\tau(c_1') = \tau(c_1)$, $|c_1'| \preccurlyeq |c_1|$ et $[\![c_1']\!](\sigma\rceil^i) = \sigma'\rceil^i$ avec $\sigma' = [\![c_1]\!](\sigma)$. Par récurrence directe sur la sémantique de la boucle *while* on conclut que la commande $while(e)\{c_1'\}$ satisfait la propriété.

□

Nous avons maintenant tous les outils nécessaires pour montrer que `TWhile` satisfait la propriété d'intégrité.

Théorème 31. *`TWhile` satisfait la propriété d'intégrité et l'équation (8.4).*

Démonstration. La politique d'intégrité est obtenue par application directe de la propriété 30 à la sémantique d'un programme.

Par la propriété d'intégrité un programme de type i ne peut modifier que des données de niveaux inférieurs. Comme pour exécuter un programme il faut qu'il réside sur un niveau inférieur à sont type, on conclut que l'équation (8.4) est satisfaite. □

8.5 Conclusions

Nous avons mis en évidence deux stratégies de protection. La première utilise la compression des programmes afin de limiter l'espace disponible à un virus pour s'introduire à l'intérieur d'un hôte. Cette approche répond tout à fait aux techniques d'infection usuelles. En effet, la plupart d'entre elles recherchent des espaces libres pour insérer un code viral. Par exemple, les espaces permettant d'aligner les sections d'un programme avec les pages mémoire du système ou ceux positionnant les pointeurs de fonction sur des adresses multiples d'une puissance de 2, sont des lieux de prédilection pour les infections. Pour plus de détails sur ces aspects pratiques nous revoyons le lecteur vers les ouvrages [Fil04, Szö05], il peut aussi consulter l'article [Kac07] où ce type d'infection est traité dans un environnement Linux.

Même si la complexité de Kolmogorov reste une fonction non calculable, il est est tout à possible d'utiliser cette première stratégie en employant une approximation de $\mathcal{K}$. En effet, les algorithmes de compression usuels sont de bonnes approximations de cette fonction comme cela est montré dans l'étude [CV05].

La seconde stratégie fait intervenir une politique garantissant l'intégrité de certaines données. En suivant les travaux [Mye99, MB05], nous avons montré que cette stratégie peut être facilement établie dans le cadre du langage de programmation `While`. Nous pensons qu'une telle initiative est un pas en avant vers la conception de systèmes immunisés.

Dans ces deux stratégies, l'argument qui nous a permis de limiter la propagation des virus fait intervenir une propriété restreignant la relation entre la sémantique et la syntaxe concrète.

Détection de programmes malicieux

9 Détection abstraite

Les travaux de Cohen [Coh88] et Adleman [Adl88] ont établi que la détection des virus informatiques est un problème indécidable. En d'autres termes, il n'existe pas de procédure effective permettant d'identifier l'ensemble des virus. Ces résultats sont obtenus par des méthodes de diagonalisation.

Dans cette partie, nous complétons ces travaux en identifiant des scénarios précis, nous en déduisons des stratégies de détection puis nous évaluons les capacités de calcul associées à la mise en place de ces stratégies.

Ce travail est fondé sur le système d'équations énoncé au chapitre 5. On rappelle que $\mathbf{v} \in \mathbb{D}$ est un virus relativement à la fonction de propagation $\mathcal{B} : \mathbb{D} \to \mathbb{D}$ et au comportement $\mathcal{F} : \mathbb{D} \to \mathbb{D}$ si le système d'équations suivant est satisfait pour tout hôte $\mathbf{p} \in \mathbb{D}$ et toute donnée $t \in \mathbb{D}$.

$$\begin{cases} [\![\mathbf{v}]\!](\mathbf{p}, t) & = [\![\mathcal{B}(\mathbf{v}, \mathbf{p})]\!](t) \\ [\![\mathbf{v}]\!](\mathbf{p}, t) & = \mathcal{F}(\mathbf{b}, \mathbf{v}, \mathbf{p}, t) \text{ avec } [\![\mathbf{b}]\!] = \mathcal{B} \end{cases} \tag{9.1}$$

Suivant l'infection informatique, le problème de la détection consiste à identifier soit les codes viraux, soit les programmes infectés. Dans le premier cas, on s'intéresse à l'ensemble des programmes $\mathbf{v} \in \mathbb{D}$ satisfaisant le système (9.1), dans le second on s'intéresse à l'ensemble des programmes de la forme $\mathcal{B}(\mathbf{v}, \mathbf{p})$ où $\mathbf{v} \in \mathbb{D}$ et $\mathcal{B} : \mathbb{D} \to \mathbb{D}$ sont solutions de ce système.

Ainsi, la détection d'une infection se réduit à décider si un programme appartient à un certain ensemble. La hiérarchie arithmétique est un outil théorique permettant d'évaluer la difficulté de ce type de problèmes. Cette hiérarchie définit des classes associées à des capacités de calcul. Un ensemble appartient à une classe si les capacités de calcul associées sont suffisantes pour le décider.

Notre étude suivra le développement suivant. Dans un premier temps, nous présentons succinctement quelques classes de la hiérarchie arithmétique. Ensuite nous énumérons plusieurs scénarios d'infection blueprint et Smith. Pour chacun d'entre eux, nous évaluons la difficulté de la détection.

9.1 Hiérarchie arithmétique

Décidabilité. La *décision* d'un ensemble $\mathbb{A} \subset \mathbb{D}$ consiste à vérifier de manière uniforme si une donnée appartient à $\mathbb{A}$. La hiérarchie arithmétique définie des classes caractérisant les capacités de calculs suffisant à la décision des ensembles. Nous

n'exposons pas la totalité de cette hiérarchie, nous nous cantonnerons aux classes les plus courantes constituées des ensembles décidables, semi-décidables et Π_2.

Définition 32 (Classe Δ_1.)**.** Un *ensemble décidable* $\mathbb{A} \subset \mathbb{D}$ admet un programme $\mathbf{p} \in \mathbb{D}$ tel que pour toute donnée $t \in \mathbb{D}$
- si $t \in \mathbb{A}$ alors $[\![\mathbf{p}]\!](t) = \circ(\bullet, \bullet)$
- sinon $[\![\mathbf{p}]\!](t) = \bullet$

On dit que le programme $\mathbf{p}$ *décide* l'ensemble $\mathbb{A}$. La classe des ensembles décidables est notée Δ_1.

Cette classe regroupe les ensembles que l'on peut décider par des capacités de calcul raisonnables. Lorsqu'un ensemble n'est pas décidable, on dit simple qu'il est *indécidable*.

Pour tout programme $\mathbf{p} \in \mathbb{D}$ et toutes données $t, u \in \mathbb{D}$, on note $\neg\big([\![\mathbf{p}]\!](t) = u\big)$ si la valeur $[\![\mathbf{p}]\!](t)$ est définie et différente de u, ou si $[\![\mathbf{p}]\!](t)$ n'est pas définie, c'est-à-dire que l'exécution du programme $\mathbf{p}$ ne termine pas sur l'entrée t.

Définition 33 (Classe Σ_1)**.** Un *ensemble semi-décidable* admet un programme $\mathbf{p} \in \mathbb{D}$ tel que pour toute donnée $t \in \mathbb{D}$
- si $t \in \mathbb{A}$ alors $[\![\mathbf{p}]\!](t) = \circ(\bullet, \bullet)$,
- si $t \notin \mathbb{A}$ alors $\neg\big([\![\mathbf{p}]\!](t) = \circ(\bullet, \bullet)\big)$.

On dit que le programme $\mathbf{p}$ *semi-décide* l'ensemble $\mathbb{A}$. La classe des ensembles semi-décidables est notée Σ_1.

Les ensembles de cette classe sont à la limite des capacités de calcul effectives. En effet, pour tout ensemble $\mathbb{A} \in \Sigma_1$, il existe un programme répondant à coup sûr si on lui présente un élément de $\mathbb{A}$. Par contre, si on lui présente un élément n'appartenant pas à cet ensemble, il est possible que l'exécution ne termine pas. En d'autres termes, on attendra en vain une réponse et on ne pourra pas savoir si élément testé appartient ou non à $\mathbb{A}$.

Définition 34 (Classe Π_2)**.** Soit $\mathbb{A} \subset \mathbb{D}$ un ensemble, $\mathbb{A}$ est un *ensemble* Π_2 si il admet un programme $\mathbf{p} \in \mathbb{D}$ tel que
- si $t \in \mathbb{A}$ alors pour toute donnée $u \in \mathbb{D}$ on a $[\![\mathbf{p}]\!](t, u) = \circ(\bullet, \bullet)$,
- si $t \notin \mathbb{A}$ alors il existe une donnée $u \in \mathbb{D}$ telle que $\neg\big([\![\mathbf{p}]\!](t, u) = \circ(\bullet, \bullet)\big)$

On dit que le programme $\mathbf{p}$ Π_2-*décide* l'ensemble $\mathbb{A}$. La classe des ensembles Π_2 est simplement notée Π_2.

Cette classe contient des ensembles dont la décision surpasse les capacités de calcul raisonnables. A priori, il faut exécuter un programme sur une infinité d'entrée pour décider un ensemble $\mathbb{A} \in \Pi_2$. Si l'une des exécutions retourne $\bullet$ alors on sait que élément testé n'appartient pas à $\mathbb{A}$. Sinon, il faut attendre la fin de toutes les exécutions pour savoir si l'élément testé y appartient. En d'autres termes, si l'élément testé n'appartient pas à $\mathbb{A}$ alors on peut espérer avoir une réponse. S'il appartient à $\mathbb{A}$ alors nous attendrons en vain une réponse.

De manière triviale, on observe que la classe des ensembles décidables est incluse dans celle des ensembles semi-décidable qui elle-même est incluse dans celle des ensembles Π_2. C'est-à-dire que $\Delta_1 \subset \Sigma_1 \subset \Pi_2$.

Réductibilité. La notion de réductibilité permet de comparer la difficulté de décision des ensembles.

Définition 35 (Réductibilité)**.** Un ensemble $\mathbb{A} \subset \mathbb{D}$ est réductible à un ensemble $\mathbb{B} \subset \mathbb{D}$, et on note $\mathbb{A} \preccurlyeq \mathbb{B}$, si il existe une fonction calculable $\mathcal{F} : \mathbb{D} \to \mathbb{D}$ telle que pour toute donnée $t \in \mathbb{D}$ on a

$$t \in \mathbb{A} \Leftrightarrow \mathcal{F}(t) \in \mathbb{B} \tag{9.2}$$

Nous illustrons la notion de réductibilité par la propriété suivante.

Propriété 36. *Soient deux ensembles* $\mathbb{A}, \mathbb{B} \subset \mathbb{D}$ *tels que* $\mathbb{A} \preccurlyeq \mathbb{B}$*. Si* $\mathbb{B} \in \Delta_1$ *alors* $\mathbb{A} \in \Delta_1$*, si* $\mathbb{B} \in \Sigma_1$ *alors* $\mathbb{A} \in \Sigma_1$ *et si* $\mathbb{B} \in \Pi_2$ *alors* $\mathbb{A} \in \Pi_2$*.*

Démonstration. Soit une fonction calculable $\mathcal{F} : \mathbb{D} \to \mathbb{D}$ satisfaisant l'équation (9.2) et un programme $\mathbf{p} \in \mathbb{D}$ calculant $\mathcal{F}$. Soit $\mathbf{q} \in \mathbb{D}$ un programme décidant $\mathbb{B}$ alors $\mathbf{p}\,;\mathbf{q}$ décide $\mathbb{A}$. En effet, pour toute donnée $t \in \mathbb{D}$ on a

$$\begin{array}{rl} t \in \mathbb{A} \Rightarrow & [\![\mathbf{p}]\!](t) \in \mathbb{B} \\ \Rightarrow & [\![\mathbf{q}]\!]([\![\mathbf{p}]\!](t)) = \circ(\bullet,\bullet) \\ \Rightarrow & [\![\mathbf{p}\,;\mathbf{q}]\!](t) = \circ(\bullet,\bullet) \end{array} \qquad \begin{array}{rl} t \notin \mathbb{A} \Rightarrow & [\![\mathbf{p}]\!](t) \notin \mathbb{B} \\ \Rightarrow & [\![\mathbf{q}]\!]([\![\mathbf{p}]\!](t)) = \bullet \\ \Rightarrow & [\![\mathbf{p}\,;\mathbf{q}]\!](t) = \bullet \end{array}$$

On montre cette propriété pour les classes Σ_1 et Π_2 en suivant un raisonnement identique. □

Complétude On observe que $\preccurlyeq$ est une relation d'ordre. On dit qu'un ensemble est complet pour une certaine classe de la hiérarchie arithmétique, si d'une part il appartient à cette classe est si d'autre part, c'est un plus grand élément de cette classe sous $\preccurlyeq$. Intuitivement, si l'on sait décider un ensemble complet pour une certaine classe alors, par réduction, on sait décider tous les ensembles de cette classe.

Définition 37 (Complétude)**.** Un ensemble $\mathbb{B} \subset \mathbb{D}$ est Δ_1-complet (resp. Σ_1-complet, Π_2-complet) s'il est Δ_1 (resp. Σ_1, Π_2) et si pour tout ensemble $\mathbb{A} \in \Delta_1$ (resp. $\mathbb{A} \in \Sigma_1$, $\mathbb{A} \in \Pi_2$) on a $\mathbb{A} \preccurlyeq \mathbb{B}$.

On remarque que tout ensemble Δ_1 est Δ_1-complet. De plus, si un ensemble complet pour une classe peut être réduit à un second ensemble $\mathbb{A} \subset \mathbb{D}$ de cette même classe alors $\mathbb{A}$ est aussi un ensemble complet par transitivité de $\preccurlyeq$. Toujours par transitivité, un ensemble Σ_1-complet n'est pas dans Δ_1. Cela vient du fait que ces deux classes sont distinctes. De même, un ensemble Π_2-complet n'est pas dans Σ_1.

9.2 Détection de code viraux.

Nous nous intéressons à la détection des infections blueprint. On rappelle que dans ce contexte, les virus sont définis par leur seul code viral et que la fonction de propagation est laissée de coté. Cela revient à considérer une forme dégénérée du système d'équations (9.1) où seule l'équation suivante est prise en compte.

$$[\![\mathbf{v}]\!](t) = \mathcal{F}(\mathbf{v}, t)$$

Virus blueprint unique. Nous commençons par un scénario très simple : nous sommes confrontés à un unique virus blueprint dont le code viral $\mathbf{v} \in \mathbb{D}$ est connue. L'équation à résoudre est figée, l'ensemble des codes viraux à décider est simplement $\{\mathbf{v}\}$. On observe que cet ensemble est décidé par le programme suivant

$$\begin{array}{l} if\ (work == \mathbf{v}) \\ \ \{work = \circ\ (\bullet, \bullet)\} \\ else \\ \ \{work = \bullet\} \end{array}$$

On conclut que la détection de ce virus est réalisable par les capacités de calcul usuelles. On peut aisément généraliser ce résultat à un nombre fini de virus blueprint.

Dans un tel scénario, les logiciels antiviraux emploient une approximation de cette stratégie de détection. Ils construisent un condensat[1] des programmes malicieux à détecter puis compare ces condensats à ceux des programmes analysés. Si deux condensats correspondent alors le programme analysé est identifié comme malicieux. Cette technique est très efficace mais peu robuste. En particulier, elle est inutilisable face à une infection polymorphe.

Virus blueprint polymorphes. L'*image* d'une fonction $\mathcal{G} : \mathbb{D} \to \mathbb{D}$ est définie par

$$\{u \mid \exists t \in \mathbb{D} : u = \mathcal{G}(t)\}$$

Propriété 38. *L'image d'une fonction semi-calculable est un ensemble semi-décidable.*

Démonstration. On se réfère [Rog67]. □

Afin de rendre leur détection plus difficile, la majorité des virus informatiques utilisent des techniques de polymorphisme. Pour étudier ce scénario nous considérons une distribution blueprint dont le générateur de codes viraux $\mathbf{dst} \in \mathbb{D}$

[1] Le terme condensat correspond au mot anglais *hash*. Par exemple, le condensat d'un programme peut être obtenu par l'intermédiaire de la célèbre fonction cryptographique MD5. Une telle fonction possède une image bornée et impose que si deux entrées sont légèrement différentes alors les sorties associées seront très différentes.

est connu. Cela revient à résoudre le système d'équations suivant.

$$\begin{cases} [\![\mathbf{v}]\!](t) = \mathcal{F}_i(\mathbf{v}, t) \\ \mathbf{v} = [\![\mathbf{dst}]\!](i) \end{cases}$$

L'ensemble des codes viraux solutions de ce système est $\mathbb{V} = \{[\![\mathbf{dst}]\!](i) \mid i \in \mathbb{D}\}$. Comme $\mathbb{V}$ est l'image de la fonction $[\![\mathbf{dst}]\!]$, la propriété 38 nous assure que $\mathbb{V} \in \Sigma_1$. Par conséquent, il existe un programme qui semi-décide $\mathbb{V}$.

Ce résultat est en demie teinte : d'un coté on est certain d'identifier tous les codes viraux, de l'autre il est possible que la procédure de détection ne termine pas lors de l'analyse d'un programme sain. Ce qui est inacceptable en pratique. De plus la propriété suivante nous montre que l'on peut faire en sorte de générer un ensemble de codes viraux indécidable, quelque soit le comportement considéré.

Propriété 39. *Soient $\{\mathcal{F}_i\}_{i\in\mathbb{D}}$ un ensemble de fonctions de $\mathbb{D}$ vers $\mathbb{D}$. Si la fonction $\mathcal{F}(u,i,t) = \mathcal{F}_i(u,t)$ est semi-calculable alors il existe une distribution blueprint relativement à $\{\mathcal{F}_i\}_{i\in\mathbb{D}}$ telle que l'ensemble des codes viraux $\{\mathbf{v} \mid \exists i : \mathbf{v} = [\![\mathbf{dst}]\!](i)\}$ est Σ_1-complet.*

Démonstration. On se munit d'une fonction de padding calculée par un programme $\mathbf{pad} \in \mathbb{D}$ et on note $\mathbf{pad}'$ le programme suivant.

$$\begin{array}{ll} [x, y] = work\,; & // \text{ décompose l'entrée} \\ work = \mathbf{univ}(x, y) & // \text{ calcule le code viral } \mathbf{v}_i = [\![\mathbf{dst}]\!](i) \\ y = \mathbf{univ}(y, y) & // \text{ calcule } [\![i]\!](i) \\ work = \mathbf{pad}(work, y) & // \text{ calcule } [\![\mathbf{pad}]\!](\mathbf{v}_i, j) \text{ avec } j = [\![i]\!](i) \end{array} \qquad (\mathbf{pad}')$$

Pour tout générateur de codes viraux $\mathbf{dst} \in \mathbb{D}$ et toute génération $i \in \mathbb{D}$, ce programme satisfait $[\![\mathbf{pad}']\!](\mathbf{dst}, i) = [\![\mathbf{pad}]\!](\mathbf{v}_i, j)$ avec $\mathbf{v}_i = [\![\mathbf{dst}]\!](i)$ et $j = [\![i]\!](i)$. On rappelle que par définition d'une fonction de padding les programmes $[\![\mathbf{pad}]\!](\mathbf{v}_i, j)$ et $\mathbf{v}_i$ calculent la même fonction.

On applique le théorème de récursion explicite à $\mathcal{G}(u,i,t) = \mathcal{F}(\mathit{spec}(\mathbf{pad}', u), i, t)$. On obtient un générateur $\mathbf{dst}$ tel que pour toute génération $i \in \mathbb{D}$ on a

$$[\![\mathbf{v}_i]\!](t) = \mathcal{F}(\mathit{spec}(\mathbf{pad}', \mathbf{dst}), i, t) \qquad \text{avec } \mathbf{v}_i = [\![\mathbf{dst}]\!](i)$$

On pose $\mathbf{dst}' = \mathit{spec}(\mathbf{pad}', \mathbf{dst})$. On observe que pour toute génération $i \in \mathbb{D}$ on a

$$\begin{array}{rll} [\![\mathbf{dst}']\!](i) &= [\![\mathbf{pad}']\!](\mathbf{dst}, i) & \text{par définition du spécialiseur} \\ &= [\![\mathbf{pad}]\!](\mathbf{v}_i, j) & \text{par définition de } \mathbf{pad}' \end{array}$$

Par conséquent on a $[\![\mathbf{v}'_i]\!](t) = \mathcal{F}(\mathbf{dst}', i, t)$ avec $\mathbf{v}'_i = [\![\mathbf{dst}']\!](i)$. On conclut que $\mathbf{dst}'$ est le générateur de code viraux d'une distribution blueprint de comportements $\{\mathcal{F}_i\}_{i\in\mathbb{D}}$.

On montre maintenant que l'ensemble des codes viraux $\mathbb{V} = \{[\![\mathbf{dst}']\!](i) \mid i \in \mathbb{D}\}$ est Σ_1-complet. Pour cela, on observe que cet ensemble peut être réduit à l'ensemble

$\mathbb{K} = \{(i, [\![i]\!](i)) \mid i \in \mathbb{D}\}$, qui est Σ_1-complet [Rog67]. En effet, prenons le programme suivant noté $\mathbf{r}$.

$$\begin{array}{l} [x, y] = work\ ; \\ work = \mathbf{dst}(i)\ ; \\ work = \mathbf{pad}(work\ , y) \end{array} \qquad (\mathbf{r})$$

Comme **dst** et **pad** calculent des fonctions totales, on conclut que $[\![\mathbf{r}]\!]$ est une fonction totale. Or pour toute donnée $i \in \mathbb{D}$ on a

$$\begin{array}{ll} [\![\mathbf{r}]\!](i, [\![i]\!](i)) = [\![\mathbf{pad}]\!](\mathbf{v}_i, j) & \text{avec } \mathbf{v}_i = [\![\mathbf{dst}]\!](i) \text{ et } j = [\![i]\!](i) \\ \qquad\qquad = [\![\mathbf{dst}']\!](i) & \end{array}$$

Par conséquent, on a $t \in \mathbb{K} \Leftrightarrow [\![\mathbf{r}]\!](t) \in \mathbb{V}$, ce qui conduit à $\mathbb{K} \preccurlyeq \mathbb{V}$. On conclut que $\mathbb{V}$ est Σ_1-complet. □

Face à ce scénario, les logiciels antiviraux ont choisi la stratégie suivante : ils identifient une signature présente dans tous les codes viraux de la distribution. Une signature est constituée d'une séquence d'octets, ou d'une expression rationnelle d'octets. La détection consiste à rechercher cette signature à travers les programmes du système, si elle est présente dans un programme alors il est détecté comme malicieux. Cette stratégie est très efficace en terme de ressources de calcul, néanmoins elle peut produire des faux positifs[2]. En effet, il est tout à fait envisageable qu'un programme inoffensif comporte la signature d'un programme malicieux. Pour le moment, les cas de faux positifs restent marginaux. Toutefois, l'accroissement perpétuel du nombre de programmes malicieux fait que ces faux positifs restent une préoccupation quotidienne.

Le germe des codes viraux. Dans le scénario précédent, la solution utilisée par les logiciel antiviraux est de détecter un sur-ensemble des codes viraux. La problématique des faux positifs nous amène à nous interroger sur l'élaboration d'une stratégie analogue n'engendrant aucun faux positif.

Dans cette optique, Adleman [Adl88] a proposé de s'intéresser à la notion de germe. Etant donné un ensemble de codes viraux $\mathbb{V} \subset \mathbb{D}$, son *germe* est l'ensemble $\mathcal{Germ}(\mathbb{V}) = \{\mathbf{p} \mid \exists \mathbf{v} \in \mathbb{V} : [\![\mathbf{p}]\!] = [\![\mathbf{v}]\!]\}$. Cet ensemble est constitué des programmes calculant la même fonction que l'un des codes viraux. Il est raisonnable de s'autoriser à détecter ces programmes puisque, même si ce ne sont pas des codes viraux, ils se comportent comme tels. On dit que $\mathbb{V}$ peut être *isolé à l'intérieur de son germe* si il existe un ensemble décidable $\mathbb{A} \in \Delta_1$ tel que

$$\mathbb{V} \subset \mathbb{A} \subset \mathcal{Germ}(\mathbb{V}) \qquad (9.3)$$

[2] On rappel qu'un faux positif est un programme inoffensif détecté par un logiciel antiviral de manière erronée.

La stratégie de détection consiste alors à rechercher les programmes de $\mathbb{A}$. On notera que cette approche s'apparente à une détection comportementale puisque nous cherchons à isoler un ensemble de programme ayant un comportement infectieux donné.

Propriété 40. *Il existe une distribution blueprint dont l'ensemble des codes viraux ne peut pas être isolé à l'intérieur de son germe.*

Démonstration. On prend le générateur $x = \mathbf{univ}(work, work)$ et on observe qu'il engendre les codes viraux $\mathbb{V} = \{[\![i]\!](i) \mid \exists i \in \mathbb{D}\}$. Par l'absurde, supposons qu'il existe un ensemble décidable $\mathbb{A} \subset \mathbb{D}$ satisfaisant l'équation (9.3). On prend $\mathbf{r} \in \mathbb{D}$ un programme décidant $\mathbb{A}$ et on note $\mathbf{r}'$ le programme suivant.

$$
\begin{array}{l}
x = \mathbf{r}(work)\,; \\
if\,(x == \bullet) \\
\quad \{\ work = \bullet\ \} \\
else \\
\quad \{\ while\,(\circ(\bullet, \bullet))\{skip\}\ \}
\end{array}
\qquad (\mathbf{r}')
$$

Pour toute donnée $t \notin \mathbb{A}$ ce programme satisfait $[\![\mathbf{r}']\!](t) = \bullet$ et pour toute donnée $t \in \mathbb{A}$ l'exécution de $[\![\mathbf{r}']\!](t)$ ne termine pas.

- Si $\mathbf{r}' \notin \mathbb{A}$ alors $[\![\mathbf{r}']\!](\mathbf{r}') = \bullet$, par conséquent $\mathbf{r}' \in \mathbb{V}$. Par hypothèse, on a $\mathbb{V} \subset \mathbb{A}$. Ce qui est une contradiction.
- Si $\mathbf{r}' \in \mathbb{A}$ alors $[\![\mathbf{r}']\!](\mathbf{r}')$ n'est pas défini. Comme $\mathbb{A} \subset \mathcal{G}erm(\mathbb{V})$, il existe un programme $\mathbf{v} \in \mathbb{V}$ tel que $[\![\mathbf{v}]\!] = [\![\mathbf{r}']\!]$. Puisque $\mathbf{v} \in \mathbb{V}$, l'exécution de $[\![\mathbf{v}]\!](\mathbf{v})$ termine et donc $[\![\mathbf{r}']\!](\mathbf{v})$ termine aussi. Par définition de $\mathbf{r}'$, on a alors $\mathbf{v} \notin \mathbb{A}$. Ce qui est une contradiction.

On conclut qu'un tel ensemble $\mathbb{A}$ n'existe pas et que $\mathbb{V}$ n'est pas isolable à l'intérieur de son germe. □

On remarque que le mécanisme d'isolement à l'intérieur du germe est proche de la notion de *séparabilité récursive* décrite dans [Rog67, Smu93]. En effet, isoler un ensemble $\mathbb{V}$ à l'intérieur de son germe revient à séparer $\mathbb{V}$ de $\{\mathbf{p} \mid \forall \mathbf{v} \in \mathbb{V} : [\![\mathbf{p}]\!] \neq [\![\mathbf{v}]\!]\}$. En consultant ces ouvrages, on observe qu'il est facile de construire des ensembles récursivement inséparables. Par conséquent, une détection sans faux positif semble difficile à réaliser, mis à part dans des cas triviaux.

Détection par comportement. L'étude [DPY05] a mis en évidence que certaines attaques modernes prennent garde de ne pas divulguer le générateur de code viraux. L'idée est de générer les mutations sur des serveurs compromis puis de diffuser ces mutations sur le réseau. Le générateur restant à l'abri sur ces serveurs.

Dans ce contexte, nous ne disposons que du comportement de l'infection pour identifier les codes viraux. D'un point de vue formel, on fixe une fonction $\mathcal{F} : \mathbb{D} \to \mathbb{D}$

et on cherche à identifier l'ensemble des programmes $\mathbf{v} \in \mathbb{D}$ satisfaisant l'équation suivante.

$$[\![\mathbf{v}]\!](t) = \mathcal{F}(\mathbf{v}, t)$$

Ce système est plus général que les précédents et par conséquent plus difficile à résoudre. Il n'est pas trivial de situer l'ensemble des solutions dans la hiérarchie arithmétique. Suivant la fonction $\mathcal{F}$, on se trouve dans l'une des classes[3] Σ_3, Σ_2 ou Π_2. Néanmoins, dans le cas où $\mathcal{F}$ est une fonction calculable la position de cet ensemble est bien connue.

Propriété 41. *Si $\mathcal{F} : \mathbb{D} \to \mathbb{D}$ est une fonction calculable alors l'ensemble des virus blueprint relativement au comportement $\mathcal{F}$ est un ensemble Π_2-complet.*

Démonstration. L'ensemble $\mathbb{A} = \{\circ(\mathbf{p}, \mathbf{q}) \mid [\![\mathbf{p}]\!] \text{ est totale et } [\![\mathbf{p}]\!] = [\![\mathbf{q}]\!]\}$ est Π_2, il est Π_2 complet puisque par projection on retrouve l'ensemble des programmes calculant une fonction totale, ensemble Π_2-complet de référence [Rog67]. Soit $\mathbf{r} \in \mathbb{D}$ calculant $\mathcal{F}$, on conclut en remarquant que $\mathbf{v} \in \mathbb{D}$ est un virus bluprint relativement à $\mathcal{F}$ si et seulement si $\circ(\mathbf{v}, \mathit{spec}(\mathbf{r}, \mathbf{v})) \in \mathbb{A}$. □

On comprend que la détection des codes viraux par leur comportement n'est pas réalisable par des capacités de calcul raisonnables. Ce résultat n'est pas étonnant dans la mesure où ce problème revient à décider si deux programmes ont la même sémantique.

Pour mettre en œuvre une stratégie approchant la détection par comportement, le domaine de la vérification de modèles semble être intéressant. En particulier le lecteur pourra consulter l'article [DPCJD07] où des techniques d'interprétation abstraite sont employées pour détecter des programmes malicieux.

9.3 Détection de formes infectées

Nous passons au mécanisme de propagation par vecteur d'infection. Ici l'objectif est d'identifier l'ensemble des programmes infectés par un virus. En suivant une démarche identique à celle de la partie précédente, nous proposons plusieurs scénarios réalistes pour lesquels nous étudions la difficulté de la détection.

Virus Smith unique. Nous commençons par un scénario très simple où nous sommes confrontés à un unique virus composé d'un code viral $\mathbf{v} \in \mathbb{D}$ et d'une fonction de propagation $\mathcal{B} : \mathbb{D} \to \mathbb{D}$, tous deux connus. Le système d'équations (9.1) est

[3] Par soucis de concision, nous n'avons pas présenté les classes Σ_3 et Σ_2. Brièvement, la classe Σ_2 est constituée des complémentaires des ensemble Π_2. La classe Σ_3 est une sur-classe de l'union $\Sigma_2 \cup \Pi_2$.

figé et il existe une unique solution. Cette solution engendre l'ensemble des formes infectées suivant.

$$\mathbb{I} = \{\mathcal{B}(\mathbf{v}, \mathbf{p}) \mid \mathbf{p} \in \mathbb{D}\}$$

On peut naturellement considérer que la fonction $\mathcal{B}$ est semi-calculable. Par la propriété 38, l'ensemble $\mathbb{I}$ est donc Σ_1. Néanmoins, quelque soit le comportement de l'infection, on peut faire en sorte que cet ensemble soit indécidable.

Propriété 42. *Pour toute fonction semi-calculable $\mathcal{F} : \mathbb{D} \to \mathbb{D}$, il existe un virus Smith de comportement $\mathcal{F}$ dont l'ensemble des formes infectées est Σ_1-complet.*

Démonstration. On suit le même développement que celui de la preuve de la propriété 39. On construit une fonction de propagation $\mathcal{B}' : \mathbb{D} \to \mathbb{D}$ de la forme $\mathit{pad}(\mathcal{B}(\mathbf{v}, \mathbf{p}), [\![\mathbf{p}]\!](\mathbf{p}))$ où $\mathbf{v} \in \mathbb{D}$ et $\mathcal{B} : \mathbb{D} \to \mathbb{D}$ sont donnés par la propriété 8. □

Germe d'une infection. Dans la partie précédente, nous avons envisagé une stratégie de détection fondée sur le germe d'un ensemble. On peut appliquer la même méthode en vue de détecter les formes infectées. Pour cela, il suffit de définir le *germe* $\mathcal{G}\mathit{erm}(\mathbb{I})$ d'un ensemble de formes infectées $\mathbb{I} \subset \mathbb{D}$ comme l'ensemble des programmes calculant la même fonction que l'une des formes infectées. C'est-à-dire

$$\mathcal{G}\mathit{erm}(\mathbb{I}) = \{\mathbf{p} \mid \exists \mathbf{q} \in \mathbb{I} : [\![\mathbf{p}]\!] = [\![\mathbf{q}]\!]\}$$

Comme précédemment, il est raisonnable de détecter les programmes du germe puisqu'ils se comportent comme des formes infectées. On dit que $\mathbb{I}$ peut être isolé à l'intérieur de son germe si il existe un ensemble décidable $\mathbb{A} \in \Delta_1$ tel que

$$\mathbb{I} \subset \mathbb{A} \subset \mathcal{G}\mathit{erm}(\mathbb{I}) \tag{9.4}$$

A l'évidence, il existe des infections permettant de contrecarrer cette stratégie de détection.

Propriété 43. *Il existe un virus tel que l'ensemble des formes infectées qu'il engendre ne peut pas être isolé à l'intérieur de son germe.*

Démonstration. On procède comme dans la preuve de la propriété 43 en posant $[\![\mathbf{v}]\!](\mathbf{p}, t) = [\![\mathit{univ}(\mathbf{p}, \mathbf{p})]\!](t)$ et $\mathcal{B}(\mathbf{v}, \mathbf{p}) = \mathit{univ}(\mathbf{p}, \mathbf{p})$. □

Distribution Smith. On suppose être confronté à une distribution Smith dont le générateur de codes viraux $\mathbf{dst} \in \mathbb{D}$ et la fonction de propagation $\mathcal{B} : \mathbb{D} \to \mathbb{D}$ sont connus. Dans le cadre du système (9.1), on fixe $\mathbf{v} = [\![\mathbf{dst}]\!](i)$ ainsi que la fonction de propagation $\mathcal{B}$. L'ensemble des formes infectées engendrées par les solutions de ce système est

$$\mathbb{I} = \{\mathcal{B}(\mathbf{v}, \mathbf{p}) \mid \mathbf{p} \in \mathbb{D} \text{ et } \exists i \in \mathbb{D} : \mathbf{v} = [\![\mathbf{dst}]\!](i)\}$$

Cet ensemble peut aussi être vu comme l'image de la fonction semi-calculable $\mathcal{G}(i, \mathbf{p}) = \mathcal{B}(\mathit{univ}(\mathbf{dst}, i), \mathbf{p})$, il est donc Σ_1. Néanmoins, il est facile de construire une distribution Smith telle que l'ensemble $\mathbb{I}$ soit Σ_1-complet, et ce quelque soit le comportement considéré.

Propriété 44. *Soient $\{\mathcal{F}_i\}_{i\in\mathbb{D}}$ un ensemble de fonctions de $\mathbb{D}$ vers $\mathbb{D}$. Si la fonction $\mathcal{F}(w, u, i, t) = \mathcal{F}_i(w, u, t)$ est semi-calculable alors il existe une distribution Smith relativement à $\{\mathcal{F}_i\}_{i\in\mathbb{D}}$ telle que l'ensemble des formes infectées engendrées est Σ_1-complet.*

Démonstration. Il suffit de reprendre la preuve de la propriété 39 ou celle de la propriété 42. □

Détection du mécanisme de propagation. Nous considérons un scénario où une faille de sécurité permettant la propagation de virus informatiques n'est pas encore corrigée. Nous désirons identifier l'ensemble des programmes se propageant par l'intermédiaire de cette faille.

Dans notre formalisme, la faille est modélisée par une fonction de propagation $\mathcal{B} : \mathbb{D} \to \mathbb{D}$. Du point de vue du système d'équations (9.1), cela revient à fixer la fonction $\mathcal{B}$ et à étudier l'ensemble des solutions possibles. En d'autres termes, on cherche l'ensemble des programmes de la forme $\mathcal{B}(\mathbf{v}, \mathbf{p})$ où $\mathbf{v} \in \mathbb{D}$ est solution du système. Il est difficile de situer cet ensemble dans la hiérarchie arithmétique. Néanmoins, on peut exhiber des fonctions de propagation pour lesquelles il est Π_2-complet.

Propriété 45. *Il existe une fonction de propagation telle que l'ensemble des formes infectées pour un certain code viral est Π_2-complet.*

Démonstration. Il suffit de prendre $\mathcal{B}(\mathbf{v}, \mathbf{p}) = \mathit{spec}(\mathbf{nil}, \mathbf{p})$ où $\mathbf{nil} \in \mathbb{D}$ est un programme calculant la fonction constante égale à $\bullet$. Le système (9.1) nous assure que $\mathbf{v} \in \mathbb{D}$ est un virus relativement à $\mathcal{B}$ si et seulement si $[\![\mathbf{v}]\!] = [\![\mathbf{nil}]\!]$. L'ensemble des programmes calculant la fonction $[\![\mathbf{nil}]\!]$ est Π_2-complet [Rog67]. Par conséquent, $\{(\mathbf{v}, \mathbf{p}) \mid [\![\mathbf{v}]\!] = [\![\mathbf{nil}]\!] \text{ et } \mathbf{p} \in \mathbb{D}\}$ est Π_2-complet et $\{\mathcal{B}(\mathbf{v}, \mathbf{p}) \mid [\![\mathbf{v}]\!] = [\![\mathbf{nil}]\!] \text{ et } \mathbf{p} \in \mathbb{D}\}$ l'est aussi. □

On observe que pour d'autres fonctions de propagation l'ensemble des formes infectées est décidable. Par exemple, il suffit de prendre $\mathcal{B}(\mathbf{v}, \mathbf{p}) = \mathbf{p}$ pour que tout programme soit une forme infectée. A la connaissance de l'auteur, aucune étude générale des solutions de ce type de systèmes n'a été réalisée. Un tel travail ne semble pas trivial et repose certainement sur une classification des fonctions de propagation. On rejoint la conclusion du chapitre 5 : il semble nécessaire d'affiner la notion de propagation pour continuer l'étude des virus informatiques.

Détection comportementale. Il reste un dernier scénario à envisager, celui où l'on ne connaît que le comportement de l'infection. Dans le système (9.1), cela revient à seulement fixer la fonction $\mathcal{F} : \mathbb{D} \to \mathbb{D}$, tous les autres paramètres sont libres. Nous cherchons alors à identifier l'ensemble des formes infectées $\mathcal{B}(\mathbf{v}, \mathbf{p})$ où le code viral $\mathbf{v} \in \mathbb{D}$ et la fonction de propagation $\mathcal{B} : \mathbb{D} \to \mathbb{D}$ vérifient le système d'équations. Comme pour le scénario précédent, l'étude d'un tel ensemble de solutions est encore préliminaire.

9.4 Conclusions

Au long de ce chapitre nous avons vu que l'étude du système d'équations présenté au chapitre 5 permet d'évaluer la difficulté de la détection virale. En considérant différentes formes de ce système nous avons observé que le problème de la détection est parfois indécidable. En étudiant la notion de germe, nous avons établi que même une détection approchée peut aussi être un problème indécidable.

Toutefois, il ne faut pas se focaliser sur ces aspects négatifs, en effet ce chapitre ouvre plusieurs pistes de recherche. Premièrement, puisqu'une approximation sans faux positif n'est pas réalisable, il semble pertinent d'étudier des stratégies de détection dont les faux positifs sont contrôlés. Peu d'études théoriques traitent de ce sujet difficile. Par ailleurs, nous avons constat que suivant les fonctions de propagation considérées, la difficulté de la détection peut fortement varier. Or l'étude de cette notion est encore à l'état embryonnaire.

10 Détection morphologique

Pour savoir si un programme est malveillant, il est à priori nécessaire d'identifier sa sémantique. L'équivalence sémantique des programmes étant un problème indécidable, la détection des programmes malicieux semble impossible. Malgré cet état de fait, il n'est pas acceptable de simplement ignorer le problème.

La majeure partie des logiciels antiviraux se fonde sur une détection syntaxique. Le principe est d'identifier tout programme malicieux par une signature constituée d'une séquence d'octets ou d'une expression rationnelle. La détection consiste alors à rechercher ces signatures parmi les fichiers du système analysé.

Cette stratégie possède trois défauts. Premièrement, elle peut facilement être contournée. Il suffit pour cela de modifier certains des octets d'un ancien programme malicieux sans altérer son comportement. On obtient un nouveau programme non détecté. De plus cette mutation peut être réalisée de manière complètement automatique [Fil06b, CJ04]. Deuxièmement, la construction d'une signature depuis un programme malicieux est très coûteuse en temps et en ressources humaines. Il faut parfois plusieurs jours pour mettre au point une signature adéquate. Durant cette période le programme malicieux se propage librement. Le cas du vers Sapphire / Slammer a montré qu'il suffit de quelques secondes pour qu'une attaque paralyse réseau mondial. Nous sommes donc confronté à une échelle de temps avantageant les attaquants. Troisièmement, à mesure que le nombre de signatures augmente, la quantité de faux positifs devient un aspect critique. On rappelle qu'un faux positif est un programme inoffensif détecté de manière erronée.

Face à ces limites, la communauté a proposé de ne plus se limiter à une détection syntaxique mais de prendre en compte la sémantique des programmes. En suivant cette voie, nous proposons dans [BKM08] une stratégie à mi chemin entre la détection syntaxique et la détection sémantique. Nous nommons cette approche *détection morphologique*, l'idée étant de reconnaître la forme des programmes malicieux. A l'inverse de la détection syntaxique, nous ne considérons pas un programme comme une simple chaîne de caractères. Nous prenons en compte les liens sémantiques entre les différentes parties du programme afin d'obtenir un objet de plus grande dimension. Pour le moment, notre étude se limite au flot de contrôle mais nous pensons ajouter d'autres critères sémantiques par la suite.

De manière pragmatique, nous conservons l'architecture de la détection par signature, mais les signatures prennent la forme de Graphes de Flot de Contrôle (GFC). Le principal avantage de cette approche est de travailler sur des objets plus abstrait que les séquences d'octets. Par conséquent notre stratégie de détection est plus ro-

buste vis-à-vis des mutations syntaxiques. De plus, en considérant des objets de plus haut niveau, nous simplifions la tâche de construction des signatures.

Notre approche est inspirée des travaux [BMLM06], néanmoins notre mode opératoire est différent. Nous cherchons à établir une méthode efficace : nous présentons un détecteur travaillant en temps cubique et nous soupçonnons l'existence d'un algorithme de complexité inférieure. De plus nous utilisons un automate d'arbres, ainsi nous bénéficions de la propriété de Myhill-Nerode qui garantit une efficacité optimale relativement à la représentation des signatures.

Afin de valider notre approche nous avons entrepris des expériences sur des échantillons de tailles moyennes. Les résultats sont prometteurs : avec une procédure automatique de construction des signatures nous obtenons un taux de faux positifs inférieur à 0.1%. L'architecture du logiciel résultant est résumée par la figure 10.1.

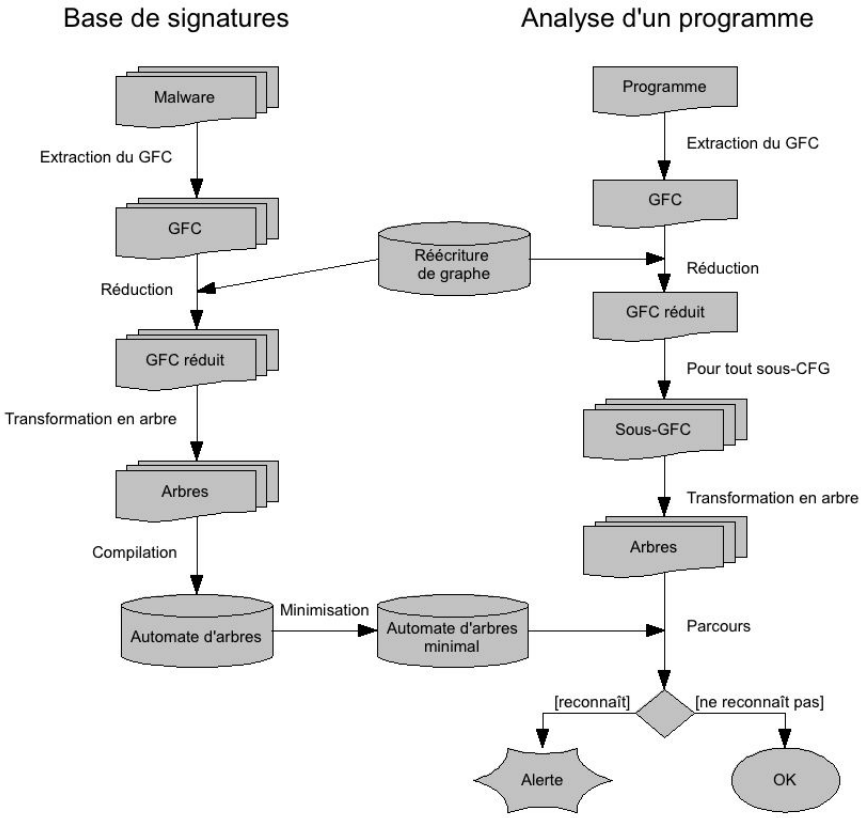


Fig. 10.1: Architecture du détecteur morphologique

Les fondements théoriques de ce chapitre sont volontairement allégés afin de mettre en avant son aspect pratique. En réalité, les expériences que nous présentons ont conduit à un approfondissement théorique conséquent en matière d'automates de graphes. L'auteur a jugé ce travail trop jeune pour être exposé dans ce manuscrit. Néanmoins cette étude complémentaire devrait prochainement donner lieu à publi-

cation. En quelques sortes, ce chapitre annonce une voie de recherche qui pourrait être entreprise à la suite de cette thèse.

10.1 GFC en assembleur x86

Nous prenons un langage assembleur simplifié. Nous exposons comment extraire un GFC depuis les instructions d'un programme en soulignant les principales difficultées rencontrées. Nous montrons aussi comment normaliser un GFC afin d'éliminer certaines mutations.

Langage assembleur. En vue d'appliquer directement notre étude à des exemples concrets, nous avons choisi de rester proche de l'assembleur x86. Le langage que nous considérons est décrit par la grammaire suivante.

$$
\begin{array}{rlll}
\text{Adresses} & \mathbb{N} & & \\
\text{Décalages} & \mathbb{Z} & & \\
\text{Registres} & \mathbb{R} & & \\
\text{Expressions} & \mathbb{E} & ::= & \mathbb{Z} \mid \mathbb{N} \mid \mathbb{R} \mid [\mathbb{N}] \mid [\mathbb{R}] \\
\text{Instructions de Flot} & \mathbb{I}^f & ::= & \mathit{jmp}\ \mathbb{E} \mid \mathit{call}\ \mathbb{E} \mid \mathit{ret} \mid \mathit{jcc}\ \mathbb{Z} \\
\text{Instructions séquentielles} & \mathbb{I}^d & ::= & \mathit{mov}\ \mathbb{E}\ \mathbb{E} \mid \mathit{comp}\ \mathbb{E}\ \mathbb{E} \mid \cdots \\
\text{Programmes} & \mathbb{P} & ::= & \mathbb{I}^d \mid \mathbb{I}^f \mid \mathbb{P}\,;\mathbb{P}
\end{array}
$$

Un programme est une séquence d'instructions $\mathbf{p} = \mathbf{i}_0\,;\cdots;\mathbf{i}_{n-1}$. L'adresse de l'instruction $\mathbf{i}_k$ est l'entier k. Ce langage ne possède que quatre types d'instructions permettant de modifier le flot de contrôle.

- Un saut inconditionnel *jmp* e transfert le contrôle vers l'adresse donnée par la valeur de l'expression e.
- Un saut conditionnel *jcc* x a deux comportements possibles. Si la condition associée est validée, le contrôle est transféré vers l'adresse obtenue par la sommation de l'adresse courante et du décalage x. Sinon le contrôle est transféré vers l'adresse de l'instruction suivante.
- Un appel de fonction *call* e empile l'adresse de l'instruction suivante puis transfert le contrôle vers l'adresse donnée par la valeur de l'expression e.
- Un retour de fonction dépile une adresse et transfert le contrôle vers cette adresse.

Prérequis. L'extraction du GFC d'un programmes est confrontée à plusieurs difficultées. Premièrement, nous avons besoin d'accéder aux instructions du programmes, ainsi les techniques d'empaquetage et de chiffrement peuvent compromettre l'extraction. Ce problème fait parti des problèmes bien connus de la discipline, en effet la détection syntaxique est aussi soumise à cette limitation. La présentation des

solutions employées par la communauté excède le cadre de cette étude. Toutefois, le lecteur pourra se référer aux ouvrages [Fil05, Fil06a, Szö05] pour plus de détails.

Deuxièmement, certaines séquences d'instructions ont une sémantique obscure. Par exemple la séquence *push a ; ret* se comporte comme le saut inconditionnel *jmp a*. Ce genre d'équivalences sémantiques est bien connu dans le domaine. Nous considérons que ces séquences sont normalisées lors de la phase de désassemblage.

Enfin, les adresses cibles des sauts et des appels de fonctions sont parfois calculées dynamiquement. Par exemple lorsque nous rencontrons le saut inconditionnel *jmp eax* nous avons besoins de la valeur du registre *eax* pour suivre le transfert de contrôle. Pour résoudre se problème, on se munit d'une heuristique $(\!|\ |\!)$ qui donne la valeur des expressions lorsqu'elles peut être calculée de manière statique. Si la valeur d'une expression e ne peut pas être calculée alors on pose $(\!|e|\!) = \bot$. Une telle heuristique peut être fondée sur la plupart des techniques d'analyse statique comme l'évaluation partielle ou l'émulation.

La procédure d'extraction. Le flot de contrôle est constitué des différents chemins pouvant être empruntés durant l'exécution d'un programme. On représente ce flot par un graphe, le GFC. Les sommets de ce graphe représentent les adresses des instructions visitées et ses arrêtes représentent les transferts de contrôle possibles entre les adresses. Le tableau 10.1 expose la procédure d'extraction d'un GFC depuis un programme.

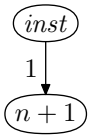
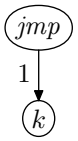
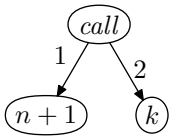
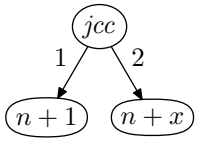
Normalisation des mutations. Notre représentation du flot de contrôle effectue une première abstraction des programmes : on ne fait aucune distinction entre les différentes instructions séquentielles. Elle sont toutes représentées par le même symbole *inst*. Cette approche rend le GFC robuste vis-à-vis des mutations simples comme la substitution d'une instruction par une autre de sémantique équivalente.

Nous renforçons encore cette robustesse par un procédé de normalisation. Certaines mutations classiques sont capables d'altérer le GFC. Nous les traitons par l'application de règles de réécriture de graphes. Nous présentons trois règles permettant de normaliser plusieurs mutation bien connues. Bien sûr, d'autre mutations existent, mais nous pensons que la plupart peut être traitée par ce principe de normalisation.

Nous utilisons les réductions suivantes.

- Concaténation des instructions séquentielles consécutives en blocs d'instructions.
- Réalignement du code par élimination des sauts inconditionnels superflus.
- Fusions des sauts conditionnels consécutifs.

Ces réductions sont modélisées par les règles de réécriture présentées par le tableau 10.2. Le tableau 10.3 expose plusieurs mutations d'un même programme, nous observons que tous les programmes obtenus ont le même GFC normalisé.

Instruction	GFC
$\mathbf{i}_n \in \mathbb{I}^d$	
$\mathbf{i}_n = jmp\ e$ $(\!\|e\|\!) = k$	
$\mathbf{i}_n = call\ e$ $(\!\|e\|\!) = k$	
$\mathbf{i}_n = jcc\ x$	
Sinon	

Tab. 10.1: Procédure d'extraction du GFC

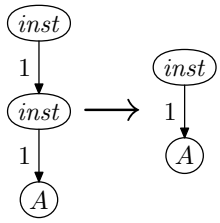
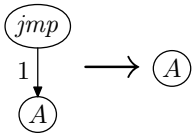
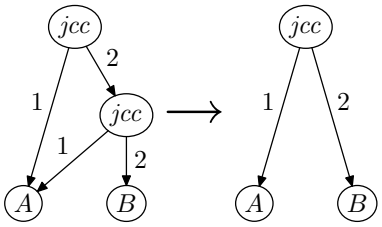
Les règles de réécriture que nous employons induisent une diminution de la taille du GFC. Ainsi la normalisation termine de manière triviale. Concernant la confluence des règles, on observe qu'il n'y a aucune paire critique. Néanmoins, dans un cas plus général, les propriétés de terminaison et de confluence devront être étudiées précautionneusement.

10.2 Une base de signatures efficace

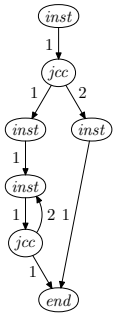
La détection morphologique est fondée sur un ensemble de GFC jouant le rôle de signatures. Ces signatures sont compilées en un automate d'arbres par l'intermédiaire d'une représentation des GFC par des arbres. Puisqu'un automate d'arbres satisfait la propriété de minimisation, nous obtenons une représentation efficace de la base de signatures. Nous appliquons ce formalisme à l'identification d'un sous-GFC afin de détecter les infections.

Dès graphes aux arbres. Un chemin est un mot sur $\{1,2\}^*$, on note ϵ le mot vide. Pour tout chemin $\rho, \mu \in \{1,2\}^*$ et tout entier $i \in \{1,2\}$, on définit l'ordre lexicographique suivant.

$$\rho 1 < \rho 2 \qquad \rho < \rho i \qquad \rho < \mu \Rightarrow \rho\rho' < \mu\mu'$$

Concaténation des instructions	Réalignement des sauts inconditionnels	Fusion des sauts conditionnels
		

Tab. 10.2: Normalisation du graphe de flot de contrôle

Programme original	Substitution d'instructions	Substitution de blocs d'instructions	Permutation de blocs d'instructions	Offuscation de sauts conditionnels	applications de toutes les mutations	GFC normalisé
0 : cmp eax 0 1 : jne +7 2 : mov ecx eax 3 : dec ecx 4 : mul eax ecx 5 : cmp ecx 1 6 : jne -3 7 : jmp +2 8 : inc ecx 9 : ret	0 : cmp eax 0 1 : jne +7 2 : mov ecx eax 3 : *sub ecx 1* 4 : mul eax ecx 5 : cmp ecx 1 6 : jne -3 7 : jmp +2 8 : *add ecx 1* 9 : ret	0 : cmp eax 0 1 : jne +8 2 : *push eax* 3 : *pop ecx* 4 : dec ecx 5 : mul eax ecx 6 : cmp ecx 1 7 : jne -3 8 : jmp +2 9 : inc ecx 10 : ret	0 : cmp eax 0 1 : jne +7 2 : mov ecx eax 3 : dec ecx 4 : mul eax ecx 5 : cmp ecx 1 6 : jne -3 9 : *ret* 8 : inc ecx 9 : *jmp -2*	0 : cmp eax 0 1 : jne +9 2 : mov ecx eax 3 : dec ecx 4 : mul eax ecx 5 : cmp ecx 2 6 : *ja -3* 7 : cmp ecx 1 8 : *jne -5* 9 : jmp +2 10 : inc ecx 11 : ret	0 : cmp eax 0 1 : *je +2* 2 : *jmp +10* 2 : *push eax* 3 : *pop ecx* 4 : *sub ecx 1* 5 : mul eax ecx 6 : *cmp ecx 2* 7 : *ja -3* 8 : cmp ecx 1 9 : *jne -5* 10 : *ret* 11 : *add ecx 1* 12 : *jmp -2*	

Tab. 10.3: Mutation du GFC

Un domaine arborescent est un ensemble $\mathbb{T} \subset \{1,2\}^*$ tel que pour tout chemin $\rho \in \{1,2\}^*$ et tout entier $i \in \{1,2\}$ on a

$$\rho i \in \mathbb{T} \Rightarrow \rho \in \mathbb{T}$$

Un arbre sur un ensemble de symboles Σ est une paire $t = (\mathbb{T}, \hat{t})$ où $\mathbb{T}$ est un domaine arborescent et $\hat{t}$ est une fonction de $\mathbb{T}$ vers Σ.

On prend l'ensemble de symboles $\Sigma = \{inst, jmp, call, jcc, ret\} \cup \{1,2\}^*$. Les feuilles étiquetées par un chemin $\rho \in \{1,2\}^*$ est vu comme un pointeur vers le sommet du domaine arborescentρ. Ainsi, il y a deux types de sommets : les sommets intérieurs étiquetés par l'un des symboles de $\{inst, jmp, call, jcc, ret\}$ et les pointeurs étiquetés par l'un des chemins du domaine arborescent. Pour tout arbre $t = (\mathbb{T}, \hat{t})$, on note $\mathring{\mathbb{T}}$ l'ensemble des sommets intérieurs, c'est à dire

$$\mathring{\mathbb{T}} = \left\{ \rho \,\middle|\, \rho \in \mathbb{T} \text{ et } \hat{t}(\rho) \in \{inst, jmp, call, jcc, ret\} \right\}$$

Un arbre $t = (\mathbb{T}, \hat{t})$ est bien formé si tous ses pointeurs pointent vers un sommet intérieur et lexicographiquement inférieur. C'est à dire que pour tous chemins $\rho, \mu \in \mathbb{T}$ on a

$$\left(\hat{t}(\rho) = \mu\right) \Rightarrow \left(\mu \in \mathring{\mathbb{T}} \text{ et } \rho \leq \mu\right)$$

On observe que tout GFC peut être représenté par un unique arbre bien formé. Cet arbre est composé d'un nombre de sommets au plus deux fois supérieur au nombre de sommets du GFC. De plus la longueur des chemins utilisés comme pointeurs est inférieure à la profondeur du GFC qui est elle même inférieure au nombre sommets du GFC. On conclut que tout GFC peut être représenté par un arbre de taille inférieure au carré de la taille du GFC.

Automates d'arbres. Un *automate d'arbres* est un quadruplet $\mathcal{A} = (\mathbb{Q}, \Sigma, \mathbb{Q}_f, \Delta)$ où $\mathbb{Q}$ est un ensemble fini d'états, Σ est un ensemble de symboles, $\mathbb{Q}_f \subset \mathbb{Q}$ est un ensemble d'états finaux et Δ est un ensemble fini de règles de transitions de la forme $a(q_1, q_2) \rightarrow q$ avec $a \in \Sigma$ un symbole d'arité i et $q, q_1, q_2 \in \mathbb{Q}$. La taille d'un automate d'arbres est défini comme le nombre de ses règles.

Le calcul du parcours d'un automate d'arbres sur un arbre $t = (\mathbb{T}, \hat{t})$ consiste à étiqueter les sommets de $\mathbb{T}$ en commençant par les feuilles en remontant vers la racine selon le principe suivant. Pour tout sommet $\rho \in \mathbb{T}$ tel que les sommets $\rho 1, \rho 2$ sont respectivement étiquetés par les états $q_1, q_2 \in \mathbb{Q}$. Si $a(q_1, q_2) \rightarrow q$ est une règle de l'automate alors le sommet ρ est étiqueté par l'état q. Etant donné un automate fixé, le parcours sur un arbre t se calcule en temps linéaire[1].

Un arbre t est *accepté* par un automate d'arbres s'il existe un parcours tel que le sommet ϵ soit étiqueté par un état final de $\mathbb{Q}_f$. Le langage reconnu par un automate d'arbres est constitué de l'ensemble des arbres acceptés par cet automate.

Une propriété fondamentale est que tout automate d'arbres peut être transformé en un automate minimal en son nombre d'états. Cet automate est minimal dans le sens ou tout automate reconnaissant le même langage comporte plus d'états. Par conséquent, l'automate minimal constitue la meilleur représentation, en termes d'automates d'arbres, du langage reconnu. L'opération de construction d'un automate minimal depuis un automate quelconque est appelée *minimisation*, elle s'effectue en un temps quadratique[2] en la taille de l'automate à minimiser. Pour plus de détails concernant cette propriété on se référera à l'ouvrage de référence [CDG+97].

Construction de la base de signatures. On prend l'ensemble $\{t_1, \cdots, t_n\}$ des arbres représentant les GFC des programmes malicieux connus. Comme cet ensemble d'arbres est fini, il existe un automate le reconnaissant. Cet automate représente notre base de signatures. Afin d'optimiser cet base on peut alors procéder à la minimisation de l'automate. Nous obtenons un automate minimal $\mathcal{A}$ reconnaissant l'ensemble des arbres représentant le GFC d'un programme malicieux.

Lorsqu'un programme malicieux infecte un autre programme, il inclut son code à l'intérieur de son hôte. On peut raisonnablement penser que le GFC du programme malicieux apparaît comme sous-graphe du GFC de l'hôte infecté.

[1] $O(n)$ avec n la taille de l'arbre à reconnaître.
[2] $O(n^2)$ où n est la taille de l'automate à minimiser.

Sous cette hypothèse, pour détecter si un programme est infecté, il suffit de prendre l'ensemble de ses sous-GFC et d'utiliser l'automate $\mathcal{A}$ pour savoir si l'un d'eux est identique au GFC d'un programme malicieux connu. Si l'un de ces sous-GFC est reconnu par l'automate on signale le programme analysé comme malicieux, sinon le programme est signalé comme sain.

Nous ne sommes pas confronté au problème général de la recherche de sous-graphes, connu pour être NP-complet. D'une part, les sous-graphes à rechercher sont fixés, ainsi le problème est clairement polynômial. De plus, les sous-graphes recherchés sont des GFC, or les contraintes d'arité et d'étiquetage des arrêtes impliquent qu'un GFC ne peut accepter qu'un nombre de sous-GFC au plus égal au nombre de ses sommets.

Sachant qu'un parcours se calcule en temps linéaire et que les GFC sont représentés par des arbres de taille inférieure au carré de la taille du GFC, on conclut que le processus de détection s'effectue en temps cubique[3] en la taille du GFC du programme à analyser.

Cette architecture de la base de signatures possède plusieurs avantages. Premièrement, lorsque qu'un nouveau programme malicieux est découvert, il est facile d'ajouter sa signature à la base en calculant une union avec l'ancien automate. Cette union se calcul en temps linéaire, ainsi il est possible de fournir rapidement un automate permettant de détecter le nouveau programme malicieux. Par contre, même s'il reste fonctionnel, cette automate d'arbres n'est pas optimal puisqu'il n'est pas minimal. La minimisation est plus coûteuse que l'union puisqu'elle s'effectue en temps quadratique. C'est pour cela que durant la minimisation, il est préférable d'utiliser l'automate « non minimisé » obtenu par union. Un fois la minimisation terminée, on peut alors mettre à jour l'automate utilisé pour la détection.

Pour donner un ordre de grandeur, dans nos expériences la minimisation requière quelques dizaines de minutes alors que l'union ne requière que quelques secondes.

10.3 Expériences

Pour valider notre approche nous avons conçu un prototype du détecteur que nous venons de décrire. Afin d'évaluer la capacité à discriminer les programmes malicieux par leur GFC, nous avons testé ce prototype sur une collection de 10156 programmes malicieux de la collection VX Heavens [vxh] et une collection de 2653 programmes sains récoltée sur installation vierge de Windows Vista. Pour cela, nous nous intéressons au taux de faux positifs, c'est à dire la quantité de programmes sains détectés de manière erronée.

Extraction de GFC. Afin de dénouer les difficultés soulignés dans la partie 10.1, nous avons choisie les solutions suivantes.

[3] $O(n^3)$ avec n la taille du GFC du programme à analyser.

– Pour dépaqueter les programmes et accéder au code nous utilisons les capacités du logiciel ClamAV [cla].
– Nous utilisons une procédure de désassemblage dynamique fondée sur la bibliothèque Udis86 [Moh]. Ce module permet de suivre le flot de contrôle au fur et à mesure du désassemblage. Il garde une trace de l'état de la pile afin d'interpréter les séquences obscures telles que *push a ; ret*.
– Notre heuristique (( )) est fondée sur un module d'émulation. Lorsque nous rencontrons une instruction de flot nécessitant l'évaluation d'une expression, nous interprétons le bloc d'instructions séquentielles précédent afin de retrouver la valeur de cette expression. Dans sa version courante ce module est limité à un sous-ensemble du langage assembleur x86. Les interruptions et les appels au système ne sont pas pris en compte.
– Nous normalisons les GFC en utilisant les règles définies par le tableau 10.2.

La figure 10.2 donne les tailles de GFC obtenues depuis les collections de programmes malicieux et de programmes sains. Sur l'axe Y on a le pourcentage de programmes dont le GFC possède un nombre de sommets inférieur à la borne indiquée sur l'axe X.

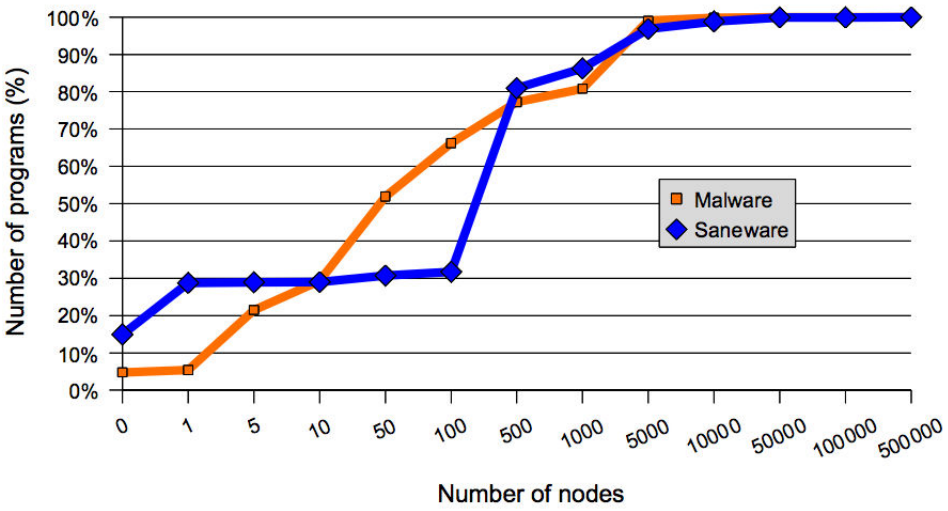


FIG. 10.2: Tailles des GFC

On observe que 65% des programmes malicieux ont un GFC avec un nombre de sommet supérieur à 15. Les 35% restant ont un GFC composé de 0 à 15 sommets. Comme nous le verrons par la suite, ces GFC sont trop petits pour constituer des signatures pertinentes. Nous travaillons actuellement sur le module d'extraction de GFC afin d'améliorer les performances de notre prototype sur cette partie de la collection.

Concernant la base de programmes sains, 30% de la collection possède un GFC composé de 0 ou 1 sommet. Ces programmes sont principalement des bibliothèques dont les points d'entrées n'ont pas pu être inférés. La majorité des programmes

restant possède un GFC de plus de 100 sommets.

Sélection des signatures. Au cours des expériences, nous avons observé qu'un faible nombre de programmes malicieux possédant de petits GFC engendre la majorité des faux positifs. Afin d'évaluer cet aspect, nous avons considéré plusieurs bornes inférieurs sur la taille des GFC entrant dans la base de signatures.

Nous obtenons le protocole expérimental suivant. Etant donnée un borne inférieure $N \in \mathbb{N}$, on construit l'automate minimal reconnaissant l'ensemble des arbres représentant les GFC de programmes malicieux composés de plus de N sommets. On définit alors le détecteur morphologique $D^N_{\mathbb{M}}$ comme le prédicat qui pour tout programme $\mathbf{p} \in \mathbb{P}$ satisfait $D^N_{\mathbb{M}}(\mathbf{p}) = 1$ si le GFC d'un programme malicieux apparaît comme sous-graphe du GFC du programme $\mathbf{p}$ et $D^N_{\mathbb{M}}(\mathbf{p}) = 0$ sinon. Ce prédicat est décidé par l'intermédiaire de l'automate d'arbres précédemment construit.

Évaluation. Nous nous intéressons aux faux positifs, on rappelle qu'ils sont constitués des programmes sains détectés comme malicieux. Nous donnons une évaluation de cet ensemble de faux positifs en utilisant les 2653 programmes sains collectés. Plus formellement, avec $\mathbb{S}$ l'ensemble de ces programmes sains, on défini l'ensemble des faux positifs engendré par le détecteur $D^N_{\mathbb{M}}$ comme suit.

$$\text{Faux positifs :} \qquad \left\{\mathbf{p} \mid D^N_{\mathbb{M}}(\mathbf{p}) = 1 \text{ et } \mathbf{p} \in \mathbb{S}\right\}$$

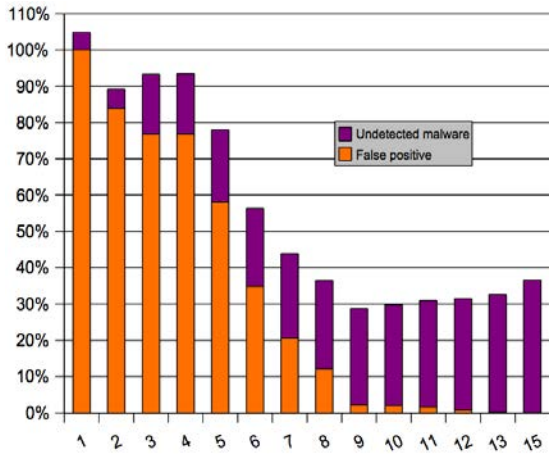
Nous n'évaluons pas les faux négatifs, c'est à dire les programmes malicieux non détecté. Nous invoquons les raisons suivantes. On ne peut utiliser notre collection pour évaluer ces faux négatif, en effet tout programme malicieux dont le GFC fait partie de la base de signature est par construction détecté. Il n'est pas non plus raisonnable d'utiliser d'autre programmes malicieux puisque un faux négatif obtenu par cette méthode serait simplement un programme ne possédant pas encore de signature. En cela nous adoptons le même point de vue que la plupart des sociétés éditrices de logiciels antiviraux.

La bonne façon de procéder serait de constituer d'une part une collection de programme infecté par notre collection de programme malicieux et d'autre part une collection de mutations résultant de l'évolution de cette même collection. A l'heure actuelle, l'auteur ne bénéficie pas des infrastructures permettant de construire de telles collections. Toutefois, la mise en place du Laboratoire de Haute Sécurité [lhs] au LORIA, devrait fournir les moyens de poursuivre ces expériences.

Néanmoins, la présence d'une borne inférieur sur la taille des GFC entrant dans la base de signatures, implique que plusieurs programmes malicieux de notre collection ne sont pas détectés. Nous mentionnons la quantité de ces programmes suivant la borne considéré. Mais on notera que ce cas de figure est plus lié au problème technique de l'extraction du GFC qu'a la méthodologie employée. Ainsi les chiffres avancés ne donnent pas une évaluation pertinente du taux de faux négatifs relatifs à la détection morphologique.

Résultats expérimentaux. Nous avons évalué le détecteur morphologique sur une collection de 2653 programmes sains. Il faut 5 h 30 min pour analyser la totalité de ces programmes. Le tableau 10.4 présente les résultats obtenus. La première colonne indique la borne inférieure sur la taille des GFC entrant dans la base de signatures, la deuxième colonne indique le taux de faux positifs et la troisième colonne indique le taux de programmes malicieux non détectés.

Borne	Faux positifs	Non détectés
1	100.00%	4.80%
2	83.78%	5.43%
3	76.82%	16.43%
4	76.77%	16.66%
5	57.98%	20.01%
6	34.84%	21.50%
7	20.57%	23.34%
9	12.06%	24.43%
10	2.17%	26.47%
11	2.04%	27.78%
12	1.60%	29.35%
13	0.71%	30.74%
15	0.09%	36.52%



Tab. 10.4: Résultats expérimentaux

Comme on pouvais le soupçonner, le taux de faux positifs diminue avec l'accroissement de la borne inférieure sur la taille des GFC. Ainsi au delà de 15 sommets, le GFC semble être un critère pertinent pour discriminer les programmes malicieux des programmes sains. D'un autre coté, l'accroissement de la borne induit une augmentation du nombre de programmes malicieux non détectés, c'est à dire ceux dont le GFC est composé d'un nombre de sommets inférieur à la borne.

Comparaisons. A titre de comparaisons, les méthodes statistiques expérimentées dans [KA94] induisent un taux de faux positifs compris entre 0.5 % et 34 %. La stratégie de détection fondée sur un réseau de neurone développé par les laboratoires IBM [TKS96] présente un taux de faux positifs inférieur à 1 %. Les méthodes de fouille de données utilisées dans [SEZS01] induisent un taux de faux positifs compris entre 2.2 % et 47.5 %. Les méthodes heuristiques commerciales testées dans [Gry99] engendrent un taux de faux positifs inférieur à 0.2 %.

10.4 Conclusions

Ce chapitre montre la faisabilité d'une détection fondée sur des propriétés sémantiques et il ouvre un nouvel axe de recherche : la détection morphologique. Ce sujet

comporte trois thèmes principaux. (a) L'extraction de GFC qui met en jeu des méthodes d'analyse statique. (b) La normalisation des GFC qui utilise des techniques de réécriture. (c) La recherche de sous-graphes qui repose sur la théorie des automates. Pour le moment, nous n'avons pas de solution pérenne pour le thème (a). Concernant le thème (b), nous comptons nous reposer sur les capacités du logiciel TOM [tom] dont l'un des axe de développement et la réécriture des GFC [BM08]. Enfin, le thème (c) a conduit à un approfondissement en termes d'automate de graphes, l'objectif étant d'identifier les sous-graphes par le calcule d'un unique parcours d'automate. Une telle avancée réduira de manière conséquente le temps d'exécution du détecteur.

Conclusion

Formalisation de la virologie informatique. Nous avons établi un formalisme très souple permettant de décrire les notions essentielles de la virologie informatique. En particulier nous avons mis en évidence les aspects fondamentaux des infections informatiques. Une telle infection est menée par l'intermédiaire d'un code viral dont la propagation est décrite par une équation fonctionnelle. Une infection polymorphe est décrite par les mêmes équations, la seule différence est que l'infection est menées par un ensemble de solutions, chacune de ces solutions caractérisant une mutation du virus informatique. Suivant le système d'équations considérée on peut alors modéliser plusieurs mécanismes de propagation.

Le théorème de récursion prend un rôle central dans ce formalisme puisqu'il permet de construire des solutions des systèmes d'équations proposés. Ce théorème s'apparente à un compilateur de virus informatiques, une dimension constructive qui lui manquait encore. Nous avons remarqué que les différents énoncés du théorème de récursion répondent aux systèmes d'équations établis. Cela suggère une taxinomie des infections informatiques que nous avons entamé en exhibant les mécanismes de reproduction blueprint et Smith. Ces deux classes permettent de modéliser des comportements symptomatiques. Le premier se focalise sur la duplication d'un code viral et le second sur la propagation d'une infection.

Plusieurs points de ce formalisme restent encore à éclairer. En particulier la notion de fonction de propagation est rudimentaire. Nous avons observé qu'elle entretient un lien privilégié avec la notion de spécialisation, néanmoins elle semble plus générale que cette dernière. La caractérisation d'un ensemble de fonctions de propagation raisonnables est une piste de recherche intéressante mais aussi très ardue. Une classification des comportements analogue à celle établie par Adleman pourrait permettre l'identification de classes de fonctions de propagation.

Par ailleurs, nous avons mis en évidence un lien entre les moteurs de mutation et les fonctions de padding dont les fondements ne sont pas encore étudiés.

Protection contre l'auto-reproduction. Nous avons adressé une des interrogations de Cohen en montrant que les capacités de calcul d'un système ne semble pas influer sur l'existence de virus informatiques. Les briques élémentaires des constructions auto-référentes étant un spécialiseur et la possibilité de composer des bloques d'instructions, il semble difficile de limiter les fonctionnalités d'un système informatique au point d'éradiquer les virus.

Néanmoins, nous avons été intrigué par le fait que certaines constructions virales sont possibles dans un contexte où elles n'ont pas de sens. Nous avons alors montré que dans certains modèles de calcul, même s'il existe des virus informatiques, leur construction ne semble pas toujours réalisable par les seules capacités de ce modèle. En d'autres termes, on peut soupçonner l'existence de systèmes informatiques qui soient incapables de construire un virus. Ceci est intéressant dans le sens où si initialement le système ne possède pas de virus et que l'on n'introduit pas non plus de virus par une intervention extérieure, alors son évolution n'engendrera pas de virus. L'existence de tels systèmes est encore hypothétique mais elle reste une piste dans le domaine de la protection antivirale.

A la suite de cette étude, nous avons mis en évidence deux stratégies de protection. La première utilise la compression des programmes afin de limiter l'espace disponible à un virus pour s'introduire à l'intérieur d'un hôte. Cette approche répond tout à fait aux techniques d'infection usuelles. En effet, la plupart d'entre elles recherchent des espaces libres pour insérer un code viral. La seconde stratégie fait intervenir une politique garantissant l'intégrité de certaines données. En stratifiant les données présentes sur un système, on est alors capable de circonscrire les infections virales. Nous avons montré que cette stratégie peut être facilement établie dans le cadre du langage de programmation `While`. Ces deux stratégies font intervenir une propriété restreignant la relation entre la sémantique et la syntaxe concrète. Nous pensons que ce type de propriété est une clé pour la conception de systèmes immunisés.

Détection de programmes malicieux. Nous avons repris les systèmes d'équations présentés au chapitre 5 afin d'évaluer la difficulté de la détection virale. En considérant différents systèmes nous avons observé que le problème de la détection est parfois indécidable. En étudiant la notion de germe, nous avons établi que même une détection approchée peut aussi être un problème indécidable. Toutefois, ce chapitre ouvre plusieurs pistes de recherche. Premièrement, puisqu'une approximation sans faux positif n'est pas réalisable, il semble pertinent d'étudier des stratégies de détection dont les faux positifs sont contrôlés. Peu d'études théoriques traitent de ce sujet difficile. Par ailleurs, nous avons constaté que suivant les fonctions de propagation considérées, la difficulté de la détection peut fortement varier. Or l'étude de cette notion est encore à l'état embryonnaire.

Dans un second chapitre, nous avons montré la faisabilité d'une détection fondée sur des propriétés sémantiques et il ouvre un nouvel axe de recherche : la détection morphologique. Ce sujet comporte trois thèmes principaux. (a) L'extraction de graphes de flot de contrôle qui met en jeu des méthodes d'analyse statique. (b) La normalisation de ces graphes qui utilise des techniques de réécriture. (c) La recherche de sous-graphes qui repose sur la théorie des automates. Pour le moment, nous n'avons pas de solution pérenne pour le thème (a). Concernant le thème (b), nous comptons nous reposer sur les capacités du logiciel TOM [tom] dont l'un des

axes de développement et la réécriture des graphes de flot de contrôle [BM08]. Enfin, le thème (c) a conduit à un approfondissement en termes d'automate de graphes, l'objectif étant d'identifier les sous-graphes par le calcul d'un unique parcours d'automate.

La conception d'un tel détecteur s'appuie sur des infrastructures d'expérimentation sécurisées. Ainsi la construction du laboratoire de haute sécurité au LORIA est maintenant un élément essentiel pour la continuité de ces recherches.

Bibliographie

[Adl88] L. Adleman – « An abstract theory of computer viruses », *Advances in Cryptology – CRYPTO'88*, vol. 403, Lecture Notes in Computer Science, 1988. 1, 2, 30, 33, 38, 40, 91, 96

[BKM06] G. Bonfante, M. Kaczmarek et J.-Y. Marion – « On abstract computer virology from a recursion-theoretic perspective », *Journal in Computer Virology* **1** (2006), no. 3-4, p. 45–54. 48, 49

[BKM07] — , « A classification of viruses through recursion theorems », *CIE*, Lecture Notes in Computer Science, vol. 4497, Springer, 2007, p. 73–82. 62

[BKM08] G. Bonfante, M. Kaczmarek et J. Marion – « An implementation of morphological malware detection », *17th EICAR*, May 2008. 103

[BM08] E. Balland et P. Moreau – « Term-graph rewriting via explicit paths », *Proceedings of the 19th Conference on Rewriting Techniques and Applications, Lecture Notes in Computer Science. Springer-Verlag* (2008). 114, 117

[BMLM06] D. Bruschi, Martignoni, L. et M. Monga – « Detecting self-mutating malware using control-flow graph matching », Tech. report, Università degli Studi di Milano, September 2006. 104

[Cas74] J. Case – « Periodicity in generations of automata », *Theory of Computing Systems* **8** (1974), no. 1, p. 15–32. 49

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison et M. Tommasi – *Tree automata techniques and applications*, vol. 10, 1997. 109

[CJ04] M. Christodorescu et S. Jha – « Testing malware detectors », *ACM SIGSOFT Software Engineering Notes* **29** (2004), no. 4, p. 34–44. 57, 103

[cla] http ://www.clamav.net. 111

[Cod68] E. Codd – *Cellular automata*, 1968. 35

[Coh86] F. Cohen – « Computer viruses », Thèse, University of Southern California, January 1986. 2, 33, 35, 37

[Coh88] — , « On the implications of computer viruses and methods of defense », *Computers and Security* **7** (1988), p. 167–184. 2, 33, 37, 91

[Coh89] — , « Models of practical defenses against computer viruses », *Computers and Security* **8** (1989), no. 2, p. 149–160. 4, 67, 83

[CV05] R. Cilibrasi et P. Vitanyi – « Clustering by compression », *Information Theory, IEEE Transactions on* **51** (2005), no. 4, p. 1523–1545. 87

[DPCJD07] M. Dalla Preda, M. Christodorescu, S. Jha et S. Debray – « A Semantics-Based Approach to Malware Detection », *POPL'07*, 2007. 98

[DPY05] O. Drori, N. Pappo et D. Yachan – « New malware distribution methods threaten signature-based av », *Virus Bulletin* (2005), p. 9–11. 97

[Fil04] E. Filiol – *Les virus informatiques : théorie, pratique et applications*, Springer-Verlag France, 2004, Translation [Fil05]. 87

[Fil05] — , *Computer viruses : from theory to applications*, Springer-Verlag, 2005. 1, 30, 33, 47, 106, 120

[Fil06a] — , *Advanced viral techniques : mathematical and algorithmic aspects*, Berlin Heidelberg New York : Springer, 2006. 106

[Fil06b] E. Filiol – « Malware pattern scanning schemes secure against black-box analysis », *15th EICAR*, 2006. 57, 103

[Gry99] D. Gryaznov – « Scanners of the Year 2000 : Heuristics », *Proceedings of the 5th International Virus Bulletin* (1999). 113

[Göd31] K. Gödel – « Über formal unentscheidbare Satze der Principia Mathematica und verwandter Systeme I », *Monatshefte für Mathematik und Physik, Complex* **38** (1931), p. 173–198. 10

[HNTJ89] T. Hansen, T. Nikolajsen, J. Träff et N. Jones – « Experiments with implementations of two theoretical constructions », *Lecture Notes in Computer Science*, vol. 363, Springer Verlag, 1989, p. 119–133. 29, 30

[JGS93] N. Jones, C. Gomard et P. Sestoft – *Partial evaluation and automatic program generation*, Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1993. 23, 29

[Jon91] N. Jones – « Computer implementation and applications of kleene's S-m-n and recursive theorems », Lecture Notes in Mathematics, Logic From Computer Science (Y. N. Moschovakis, éd.), Springer Verlag, 1991, p. 243–263. 29, 30

[Jon94] — , *Constant time factors* Do *matter*, MIT Press, Cambridge, MA, USA, 1994. 30

[Jon97] — , *Computability and complexity : From a programming perspective*, MIT Press, Cambridge, MA, USA, 1997. 2, 13, 14, 25

[KA94] J. Kephart et W. Arnold – « Automatic Extraction of Computer Virus Signatures », *4th Virus Bulletin International Conference* (1994), p. 178–184. 113

[Kac05] M. Kaczmarek – *Virologie informatique*, Mémoire, Ecole nationale supérieur des Mines de Nancy - IAEM Nancy, 2005. 51

[Kac07] M. Kaczmarek – « ELF et virologie informatique », *GNU Linux Magazine France* (2007), p. 12–19. 87

[Kle52] S. Kleene – *Introduction to metamathematics*, Van Nostrand, 1952. 23, 67

[Lan84] C. Langton – « Self-reproduction in cellular automata », *Physica D : Nonlinear Phenomena* **10** (1984), no. 1-2, p. 135–144. 35

[lhs] http ://lhs.loria.fr. 112

[Lud98] M. Ludwig – *The giant black book of computer viruses*, American Eagle Publications, 1998. 43, 47

[LV97] M. Li et P. Vitányi – *An Introduction to Kolmogorov Complexity and its Application*, Springer, 1997, (Second edition). 81

[MB05] A. Matos et G. Boudol – « On declassification and the non-disclosure policy », *Proc. IEEE Computer Security Foundations Workshop* (2005), p. 226–240. 77, 87

[Moh] V. Mohan – « Udis86 », http ://udis86.sourceforge.net. 111

[Mos06] L. Moss – « Recursion theorems and self-replication via text register machine programs », *EATCS bulletin*, 2006. 13

[Mye99] A. Myers – « JFlow : practical mostly-static information flow control », *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1999), p. 228–241. 77, 87

[Odi89] P. Odiffredi – *Classical recursion theory*, North-Holland, 1989. 26, 30

[Rog58] H. Rogers – « Gödel numberings of partial recursive functions », *Journal of Symbolic Logic* **23** (1958), no. 3, p. 331–341. 25, 28, 56

[Rog67] — , *Theory of recursive functions and effective computability*, McGraw Hill, New York, 1967. 1, 24, 25, 30, 56, 94, 96, 97, 98, 100

[SEZS01] M. Schultz, E. Eskin, E. Zadok et S. Stolfo – « Data Mining Methods for Detection of New Malicious Executables », *Proceedings of the IEEE Symposium on Security and Privacy* (2001), p. 38. 113

[Smu93] R. Smullyan – *Recursion theory for metamathematics*, Oxford University Press, 1993. 72, 97

[Smu94] — , *Diagonalization and self-reference*, Oxford University Press, 1994. 9, 26, 30, 73, 74

[Szö05] P. Ször – *The art of computer virus research and defense*, Addison-Wesley Professional, 2005. 87, 106

[Tho84] K. Thompson – « Reflections on trusting trust », *Communications of the Association for Computing Machinery* **27** (1984), no. 8, p. 761–763. 43

[TKS96] G. Tesauro, J. Kephart et G. Sorkin – « Neural networks for computer virus recognition », *Expert, IEEE [see also IEEE Intelligent Systems and Their Applications]* **11** (1996), no. 4, p. 5–6. 113

[tom] http ://tom.loria.fr. 114, 116

[Usp56] V. Uspenskii – « Enumeration operators and the concept of program », *UMN* **11** (1956). 24

[vN66] J. von Neumann – *Theory of self-reproducing automata*, University of Illinois Press, Urbana, Illinois, 1966, edited and completed by A.W.Burks. 2, 3, 33, 35, 67

[vxh] http ://vx.netlux.org. 110

[ZZ04] Z. Zuo et M. Zhou – « Some further theoretical results about computer viruses », *The Computer Journal* **47** (2004), no. 6, p. 627–633. 40

INSTITUT NATIONAL
POLYTECHNIQUE
DE LORRAINE

AUTORISATION DE SOUTENANCE DE THESE
DU DOCTORAT DE L'INSTITUT NATIONAL
POLYTECHNIQUE DE LORRAINE

o0o

VU LES RAPPORTS ETABLIS PAR :
Monsieur Serge GRIGORIEFF, Professeur, LIAFA, Université Paris Diderot, Paris
Monsieur José M. FERNANDEZ, Professeur, Polytechnique Montréal, Québec

Le Président de l'Institut National Polytechnique de Lorraine, autorise :

Monsieur KACZMAREK Matthieu

NANCY BRABOIS
2, AVENUE DE LA
FORET-DE-HAYE
BOITE POSTALE 3
F - 54501
VANDŒUVRE CEDEX

à soutenir devant un jury de l'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE, une thèse intitulée :

"Dès fondements de la virologie informatique vers une immunologie formelle"

en vue de l'obtention du titre de :

DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

Spécialité : **« Informatique »**

Fait à Vandoeuvre, le 26 novembre 2008
Le Président de l'I.N.P.L.,
F. LAURENT

INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE – NANCY –

TEL. 33/03.83.59.59.59
FAX. 33/03.83.59.59.55

topmargin0

Résumé. Cette thèse aborde trois thèmes : la formalisation de la virologie informatique, l'élaboration de protections contre l'auto-reproduction et le problème de la détection des programmes malicieux.

Nous proposons une formalisation s'appuyant sur les fondements de l'informatique théorique et sur les travaux fondateurs de la discipline. Nous obtenons un formalisme souple où le théorème de récursion prend le rôle d'un compilateur de virus informatiques. Ce théorème trouve alors la place qui lui manquait encore dans la théorie de la programmation.

Ce formalisme nous fournit des bases suffisamment solides pour étudier de nouvelles stratégies de protection. Dans un premier temps nous nous intéressons aux relations qu'entretiennent auto-reproduction et capacités de calcul afin d'identifier un modèle raisonnable où l'auto-reproduction est impossible. Ensuite nous exposons deux stratégies construite sur la complexité de Kolmogorov, un outil de l'informatique théorique reliant la sémantique et la syntaxe concrète d'un langage de programmation.

Le thème de la détection comporte deux parties. La première traite de la difficulté de la détection des virus informatiques : nous identifions les classes de la hiérarchie arithmétique correspondant à différents scénarios d'infections informatiques. La seconde partie aborde des aspects plus pratiques en décrivant l'architecture d'un détecteur de programmes malicieux conçu durant cette thèse. Ce prototype utilise une détection morphologique, l'idée est de reconnaître la forme des programmes malicieux en utilisant des critères syntaxiques et sémantiques.

Mots clés : Virus informatique, formalisation, protection, détection.

Abstract. This dissertation tackles three topics : the formalization of the computer virology, the construction of protections against self-reproduction and the issue of malware detection.

We propose a formalization that is based over computer science foundations and over the founder works of the discipline. We obtain a generic framework where the recursion theorem takes a key role. This theorem is seen as computer virus compiler, this approach provides a new programming perspective.

The sound basis of this framework allows to study new protection strategies. First, we analyze the relations between the notion of self-reproduction and the computation capabilities. We aims at identifying a reasonable model of computation where self-reproduction is impossible. Then we propose two defense strategies based on the Kolmogorov complexity, a tool which relates the semantics to the concrete syntax of programming languages.

We treat the issue of malware detection in two steps. First, we study the difficulty related to the detection of several scenarios of computer infection. Second, we present a malware detector that was designed during the thesis. It is based on a morphological detection which allies syntaxical and semantical criteria to identify the shapes of malware.

Keywords : Computer virus, formal framework, defense, detection.