UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS   UFR Sciences

École Doctorale STIC

THÈSE

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice - Sophia Antipolis

Spécialité : INFORMATIQUE

présentée et soutenue par

Guillaume DUFAY

Équipe d'accueil : LEMME – INRIA Sophia-Antipolis

Vérification formelle de la plate-forme Java Card

Thèse dirigée par Gilles BARTHE

Soutenue publiquement le vendredi 5 décembre 2003
devant le jury composé de :

Mme :	Laurence	PIERRE	I3S, Université de Nice	Présidente
MM. :	Thomas	JENSEN	IRISA / CNRS	Rapporteurs
	Erik	POLL	Université de Nijmegen, Pays-Bas	
M. :	Chris	HANKIN	Imperial College, Royaume-Uni	Examinateurs
Mme :	Christine	PAULIN	Université Paris Sud	
M. :	Gilles	BARTHE	INRIA Sophia-Antipolis	Directeur

Vérification formelle de la plate-forme *Java Card*

Thèse de doctorat

Guillaume DUFAY

Décembre 2003

Remerciements

Quelle meilleure opportunité que cette thèse, qui marque l'achèvement des études universitaires, pour exprimer mes plus vifs remerciements à mes parents? Je ne leur serais probablement jamais assez reconnaissant de m'avoir toujours fait confiance et accordé les moyens d'aboutir à ce stade.

Je tiens ensuite à remercier Laurence Pierre de m'avoir fait l'honneur de présider mon Jury de thèse, Thomas Jensen et Erik Poll d'avoir accepté la tâche délicate d'être rapporteurs, Christine Paulin et Chris Hankin d'avoir bien voulu m'accorder de leur temps en faisant partie du Jury.

J'estime tout le long de mes études avoir eu beaucoup de chance dans les personnes que j'ai rencontrées. Je leur dois énormément. Gilles, dont j'ai pu apprécier durant ces années de thèse les grandes qualités humaines et scientifiques, la disponibilité malgré des activités débordantes, illustre parfaitement celles-ci. Un sans-faute encore parmi mes collègues de bureau (et devenus bien plus) : Simão, Pierre et dernièrement Tamara. Un grand merci également à tous les membres de mon équipe d'accueil (Lemme), ainsi qu'à Bernard et Line, ne serait-ce que pour l'aide et l'attention qu'ils ont pu m'apporter.

Ces trois années de thèse m'auraient paru bien insurmontables sans ma famille d'une part et les amis qui m'ont accompagné durant cette période d'autre part. Il semblera peut-être fastidieux au lecteur de les avoir énumérés mais je n'y échapperai pas tant chacun d'entre eux a compté pour moi. Parmi ces amis, je pense à ceux que j'ai pu conserver malgré l'éloignement géographique : Amadoux, Emmanuel, Emmanuelle, Marion, Nicolas, Renaud, Richard; ainsi qu'à tous ceux de cet environnement privilégié qu'est l'INRIA : Antonia, Carine, Céline, Diane, Fabrice, Ludovic, Nicolas (l'équipe des Dey-Magaud), Marc, Marie-Claude, Pascal et Valérie, Sémi. Les incroyables richesses de la Côte d'Azur (villes, villages, plaines, forêts, rochers, et bien sûr mer et montagnes), dont j'ai tant profité, me seraient apparues bien fades sans leur présence. Merci enfin aux personnes, croisées moins souvent, mais qui, en toutes occasions, m'ont offert de bons moments.

Table des matières

Table des matières **7**

1 Introduction **13**
- 1.1 Présentation de la thèse 15
- 1.2 Motivations 16
- 1.3 Plan de la thèse 17
- 1.4 Contributions 18

2 Contexte **21**
- 2.1 Les cartes à puce 21
  - 2.1.1 Présentation physique 21
  - 2.1.2 Utilisation 22
  - 2.1.3 Les cartes intelligentes 24
- 2.2 Java Card 25
  - 2.2.1 Le langage 25
  - 2.2.2 La machine virtuelle 27
  - 2.2.3 L'environnement d'exécution 28
  - 2.2.4 La vérification de bytecode 29
- 2.3 Les méthodes formelles 30
  - 2.3.1 Les techniques 31
  - 2.3.2 Les outils 32
  - 2.3.3 Une métrique de la modélisation 35

2.4 La sécurité 36
2.4.1 Problématique 36
2.4.2 État de l'art de modélisation et de la vérification Java (Card) 38
2.5 Conclusion 42

3 Formalisation de la machine virtuelle 43
3.1 Notations de Coq 44
3.2 Représentation des programmes 45
3.2.1 Le système de types 46
3.2.2 Les méthodes 47
3.2.3 Les interfaces 48
3.2.4 Les classes 49
3.2.5 Les programmes 50
3.3 Représentation de l'environnement d'exécution 51
3.3.1 Les valeurs 51
3.3.2 Les objets 52
3.3.3 Les contextes d'exécutions de méthode 53
3.3.4 Les états 54
3.4 Représentation des instructions 55
3.4.1 Sémantique de new 56
3.4.2 Sémantique de if_acmp 57
3.4.3 Sémantique de getfield 59
3.4.4 Sémantique de invokevirtual 60
3.5 Gestion des erreurs et exceptions 62
3.5.1 Les erreurs 63
3.5.2 Les exceptions 63
3.5.3 Récupération d'exceptions 64
3.6 Exécution des programmes 65
3.6.1 Conversion des programmes 66
3.6.2 Le JCVM Tools 66
3.6.3 Exemple de conversion 69
3.6.4 Utilisation des APIs 71
3.6.5 Les méthodes natives 72
3.6.6 Le paquettage `javacard.framework` 73
3.7 Conclusion 74

4 Abstractions de la machine virtuelle **77**
4.1 Méthodologie pour la vérification de propriétés 78
4.2 La machine virtuelle offensive 80
4.2.1 Représentation de l'environnement d'exécution 80
4.2.2 Représentation des instructions 81
4.3 La machine virtuelle abstraite 83
4.3.1 Représentation de l'environnement d'exécution 83
4.3.2 Représentation des instructions 85
4.4 Jakarta 89
4.4.1 Le langage de spécification 90
4.4.2 Génération d'abstractions 94
4.5 Conclusion 100

5 Vérification de bytecode **101**
5.1 Aperçu de la vérification de bytecode 102
5.1.1 Algorithme 103
5.1.2 Le problème des sous-routines 105
5.1.3 Vérification de bytecode polyvariante 106
5.1.4 Vérification légère de bytecode 108
5.1.5 Vérification de l'initialisation des objets 109
5.2 Notations de Coq 110
5.2.1 Noms qualifiés 110
5.2.2 Les modules 111
5.2.3 Quelques définitions 112
5.3 Formalisation de vérificateurs de bytecode 113
5.3.1 Abstraction de la notion de vérificateur 113
5.3.2 Instanciation paramétrique à la machine abstraite . . 116
5.3.3 Instanciation à la machine défensive 122
5.3.4 Différents paramétrages 128
5.3.5 Un vérificateur de bytecode léger 131
5.4 Conclusions 133

6 Instanciation et outils d'aide à la preuve **135**
6.1 Notations de Coq 135
6.2 Instanciation du modèle de BCV 136
6.2.1 Machine virtuelle défensive 136

6.2.2 Machine virtuelle abstraite 137
6.2.3 Ordre sur les types et sur les états 137
6.3 Raisonnement par cas dans Coq 141
6.3.1 Preuve sur les contextes 141
6.3.2 Preuve de commutation 146
6.3.3 Preuve de monotonie 151
6.4 La tactique Coq Analyze 154
6.5 Génération automatique des preuves par Spike 156
6.6 Génération automatique des preuves par Jakarta 156
6.7 Conclusion 158

7 Conclusion 161
7.1 Résultats et méthodologie 161
7.1.1 Formalisation de la plate-forme 161
7.1.2 Méthodologie pour la vérification 162
7.1.3 Quelques chiffres 162
7.2 Perspectives 163
7.2.1 Plate-forme 163
7.2.2 Vérificateur de bytecode 164
7.2.3 Outils 164
7.2.4 Sécurité 165

Bibliographie 167

Index 177

Acronymes 179

CHAPITRE 1

Introduction

Auparavant transportés par des disquettes, désormais propagés par les réseaux, les virus, chevaux de Troie et autres programmes malicieux s'installent sur les ordinateurs personnels et disposent de leurs ressources. Ces programmes profitent d'une absence de contrôle de la part d'un utilisateur mal informé, ou d'un système d'exploitation trop laxiste. Si l'information de l'utilisateur demeurera toujours imparfaite, il reste des progrès notoires à effectuer du côté du système d'exploitation. La faiblesse de conception de ces derniers permet en effet à des programmes, sans même qu'ils aient été vérifiés, d'avoir librement accès au système de fichiers et aux fonctions de pilotage des logiciels. La seule parade alors proposée aux utilisateurs est l'acquisition d'un anti-virus, avec la contrainte de devoir le mettre à jour très régulièrement sous peine d'inefficacité.

Cette prolifération de programmes malicieux est jugée très préoccupante. Pourtant, les dégâts portés sur l'ordinateur même et plus spécifiquement sur les fichiers de ses utilisateurs restent souvent très limités et dans de nombreux cas récupérables par des programmes correctifs ou des sauvegardes. Dans les cas restants où des fichiers seraient réellement perdus, et pour une utilisation personnelle de l'ordinateur, il apparaît beaucoup plus probable que ces fichiers renferment les photos de vacances ou les scores du démineur que des données dont la perte a des répercutions économiques. En revanche, appliquées au domaine des cartes à puce intelligentes, les répercutions des programmes malicieux peuvent porter beaucoup plus à conséquences.

Les cartes à puce intelligentes. Succédant aux cartes à puce classiques telles que les cartes de téléphone, les cartes de crédit, la nouvelle génération de cartes à puce intègre un environnement d'exécution capable de gérer plusieurs programmes et d'en charger de nouveaux. Ces cartes, à diffusion massive, sont destinées à renfermer des informations confidentielles et personnelles (comme des renseignements médicaux, des clés cryptographiques) mais aussi à réaliser des transactions commerciales (porte-monnaie électronique). Dans ce contexte, des programmes malicieux pourraient accéder à ces données personnelles, les modifier ou même effectuer des transactions. Par ailleurs, le fonctionnement de ce type de programme, de part la nature même de la carte (sans retour visuel), peut être indétectable pour le porteur de la carte. L'environnement d'exécution de la carte à puce doit donc être capable de répondre aux impératifs de sécurité sur lesquels les ordinateurs personnels échouent. Les réponses proposées se centrent sur la vérification des programmes avant leur exécution et sur des mécanismes de sécurité offerts par l'environnement d'exécution, qui doit alors être correctement implémenté.

La vérification de programme. Malgré la confiance apportée aux entités proposant des programmes pour cartes à puce intelligentes, seule la vérification du programme en bout de chaîne, c'est-à-dire avant son exécution sur la carte à puce, est en de mesure de garantir que le programme se comporte correctement (du point de vue innocuité). Pour réaliser cette vérification, des recherches récentes ont permis de placer au sein même de la carte à puce des *vérificateurs de bytecode*, auparavant trop complexes en puissance de calcul et en espace mémoire. Ceux-ci ont la tâche d'assurer que les programmes sont bien formés et qu'ils ne tenteront pas, entre autres, d'accéder de manière illégale à la mémoire. Ils se basent pour cela principalement sur les *systèmes de types* (une abstraction des données représentées par les valeurs manipulées par le programme) qu'apportent les langages de programmation modernes et sur une *machine virtuelle* (une abstraction d'un processeur) qui exécute le programme. Il importe alors que cette machine virtuelle, partie essentielle de l'environnement d'exécution, soit elle aussi correcte.

La certification de l'environnement d'exécution. L'environnement d'exécution est responsable, comme le suggère son nom, de l'exécution des programmes et de fournir des services à ceux-ci. Parmi ces services figurent par exemple des moyens d'échange de données entre les programmes de la carte, seule possibilité offerte aux programmes pour communiquer entre eux. Disposer d'un environnement d'exécution certifié, conforme aux spécifications, permet d'assurer que l'exécution des programmes se déroulera comme attendu et que les mécanismes de sécurité inhérents à l'environnement d'exé-

cution sont respectés.

Java Card. La plate-forme *Java Card* (de Sun Microsystems) propose un de ces environnements d'exécution pour cartes à puce intelligentes. Les programmes destinés à cette plate-forme sont écrits dans le langage *Java Card*, dérivé du populaire *Java*. La complexité maîtrisée de cet environnement d'exécution rend possible sa certification, laquelle certification est encore aujourd'hui très difficilement envisageable pour les systèmes d'exploitation des ordinateurs personnels. D'autre part, le format structuré des programmes *Java Card*, opposé aux sources hétérogènes des programmes pour ordinateurs personnels, convient à leur vérification.

Ainsi, nous nous proposons dans cette thèse d'apporter une vérification formelle des principaux composants de la plate-forme *Java Card* ainsi qu'une méthodologie et des outils pour aider à cette vérification.

1.1 Présentation de la thèse

La vérification de la plate-forme *Java Card* présentée dans ce document prend place à l'intérieur du cadre formel offert par l'assistant à la preuve Coq. Ce dernier servira à la formalisation de l'environnement d'exécution *Java Card*, du vérificateur de code octet ainsi qu'aux différentes preuves sur la vérification de la plate-forme.

L'assistant à la preuve Coq met à disposition un mécanisme de spécification fonctionnelle basé sur la théorie des types. Le système de types utilisé, le Calcul des Constructions Inductives, entraîne une expressivité inégalée et permet de spécifier des notions très complexes. Il rend aussi possible, dans ce même formalisme, d'énoncer des propriétés sur ces spécifications et de réaliser leur preuve. Pour aider à la construction de ces preuves, le logiciel Coq inclut de nombreuses *tactiques* qui correspondent à des pas, élémentaires ou non, dans le déroulement des démonstrations.

L'essentiel de l'environnement d'exécution *Java Card* et l'intégralité de la machine virtuelle *Java Card* seront ainsi formalisés, en suivant les spécifications de Sun sur le comportement calculatoire et les vérifications à effectuer, dans le langage de spécification de Coq. Le style fonctionnel utilisé rend ces spécifications exécutables et permet donc de disposer de la même spécification pour l'exécution et le raisonnement. De plus, ce style reste très proche des langages de programmation classiques et donc très facilement assimilable pour des personnes externes au domaine de la vérification.

La machine virtuelle réalisée, qualifiée de *défensive* par les tests qu'elle effectue dynamiquement (ici le typage correct des valeurs), sera ensuite abs-

traite pour obtenir une machine *offensive* n'effectuant pas les tests (donc plus rapide à l'exécution) et une machine *abstraite* n'effectuant que les tests et pas les calculs. Cette dernière machine constituera la base du vérificateur de code octet.

Nous construirons ensuite des modèles de vérificateurs de code octet en se servant du système de modules de Coq. Ces vérificateurs seront paramétrés par un type d'analyse donné (parmi plusieurs proposés pour privilégier la rapidité de la vérification ou tenir compte de certaines particularités du langage *Java Card*, comme les sous-routines). On apportera la preuve qu'un programme passant le stade de la vérification de code octet n'occasionnera pas d'erreurs non dynamiques à l'exécution et qu'il peut ainsi être exécuté de manière sûre sur la machine offensive.

Le modèle présenté de vérificateur de code octet requerra un certain nombre de preuves et d'invariants sur les machines virtuelles pour être instancié. Nous apporterons alors les preuves nécessaires sur nos machines virtuelles.

Enfin, nous montrerons comment généraliser la méthodologie employée pour d'autres propriétés que le typage. Nous décrirons également des outils, tels que *Jakarta*, visant à faciliter l'application de cette méthodologie tant pour la construction des machines virtuelles que pour la réalisation des preuves associées.

1.2 Motivations

La plate-forme *Java Card* se prête de manière idéale à la formalisation. Bien qu'étant relativement complexe, car il ne s'agit pas d'un système dédié à l'expérimentation (comme trop souvent) mais bien d'une technologie complète et destinée sous sa forme actuelle à une large diffusion, les outils de preuve permettent maintenant d'envisager une telle formalisation. Cette thèse en apporte l'évidence.

Les bénéfices de la modélisation de la plate-forme *Java Card* se situent aussi bien du côté de la confiance accrue apportée à cette plate-forme que dans le développement d'outils génériques pour pouvoir raisonner sur un système de cette taille.

Sur la confiance apportée à la plate-forme, cette dernière demande par nature un niveau de sécurité maximum. Les applications qui doivent y fonctionner manipulent en effet des données confidentielles (dossier médical par exemple) ou effectuent des transactions à valeur économique (telles qu'un porte-monnaie électronique). Seul un modèle formel et précis est capable de capturer toutes les notions du système et permet ensuite d'en vérifier ses pro-

priétés. Notons alors qu'un modèle qui soit de plus exécutable garantit que l'exécution préserve les propriétés observées. Enfin, pour la reconnaissance publique de la sécurité d'un produit (telle que celle apportée par les Critères Communs, cf section 2.3.3) l'utilisation d'un modèle formel est requis pour les niveaux de sécurité maximaux.

Devant l'impossibilité de vérifier toutes les composantes de la plate-forme *Java Card*, une fois celles-ci modélisées, nous nous concentrons sur la plus essentielle d'entre elles pour la sécurité : le vérificateur de code octet qui garantit entre autres la sûreté du typage pour les programmes exécutés. Cependant la méthodologie adoptée pour assurer cette sûreté du typage est générique et peut s'adapter à la vérification d'autres types de propriétés sur la plate-forme. Cette adaptation est facilitée pour les outils développés pour raisonner sur ce type de système.

Afin de travailler sur des spécifications aussi imposantes que celles de la plate-forme *Java Card*, il est nécessaire de disposer d'outils proposant une automatisation des traitements. Dans le cas de Coq, qui constitue le socle de notre formalisation, l'automatisation reste relativement faible et il est possible d'apporter des *tactiques* (étapes de raisonnement) permettant de manipuler des spécifications fonctionnelles comme celles que nous utilisons. De même notre méthodologie repose sur la réalisation d'abstractions qui peuvent s'obtenir de manière presque systématique et ainsi être automatisées.

1.3 Plan de la thèse

Le chapitre 2 prolonge cette introduction en fixant une vue d'ensemble de la plate-forme *Java Card*, des outils liés aux méthodes formelles et de l'état de l'art de la modélisation de la plate-forme *Java Card*.

Le chapitre 3 donnera ensuite une présentation détaillée de la machine virtuelle *défensive* telle qu'elle a été formalisée en Coq et des outils liés à son exécution. À partir de la machine virtuelle ainsi réalisée seront dérivées au chapitre 4 les machines virtuelles *offensive* et *abstraite*. Nous fixerons alors la méthodologie générale et présenterons l'outil *Jakarta* destiné à automatiser cette phase d'abstraction.

Le chapitre 5 se centrera plus spécifiquement sur la vérification de code octet avec la présentation des principes généraux, l'état de l'art et la formalisation réalisée en Coq de vérificateurs de code octet modulaires et paramétrés. On apporta enfin au chapitre 6 les preuves nécessaires pour instancier le modèle de vérificateur de code octet à notre formalisation des machines virtuelles. Nous y présenterons également les outils développés pour faciliter

les preuves.

Nous fournirons enfin dans la conclusion (chapitre 7) un aperçu des réalisations de cette thèse et des perspectives de ces travaux.

Bien que la présentation de cette thèse s'organise composante par composante, la dépendance entre les chapitres suggère une lecture linéaire. Le chapitre 5 sur la vérification de code octet, autonome, peut néanmoins être lu indépendamment.

Notons enfin que nous ne consacrons pas de chapitre ou de section à une présentation globale des notations de Coq ou de l'outil *Jakarta*. Nous n'introduirons sur ces derniers, à l'intérieur des chapitres, que les notions liées au sujet traité.

1.4 Contributions

Les travaux présentés dans cette thèse sont le résultat d'une collaboration entre plusieurs membres de l'équipe Lemme de l'INRIA Sophia Antipolis. Nous soulignons ici, pour chacune des composantes présentées dans la suite de ce document, les contributions personnelles.

Formalisation des machines virtuelles. La sémantique de la machine virtuelle *Java Card* a été réalisée en collaboration avec Simão Melo de Sousa. Nous en avons ensuite assuré son suivi et développé les machines virtuelles abstraites et offensives. Le *JCVM Tools* a quant à lui été développé par Bernard Serpette mais nous avons assuré ensuite sa compatibilité avec l'évolution de nos spécifications.

Réalisation des vérificateurs de *bytecode*. Nous avons conçu le modèle de vérificateur de code octet, paramétré par le type d'analyse souhaité. Nous avons étendu ce concept de vérification jusqu'à la machine virtuelle défensive et proposé également un modèle de vérification légère de code octet (destinée à être embarquée sur une carte à puce).

Réalisation des preuves. Nous avons réalisé la presque totalité des quelques 18 000 lignes de preuves en rapport avec les machines virtuelles. Les propriétés les plus fastidieuses à démontrer portaient sur les preuves de monotonie et de validation croisée présentées au chapitre 6.

Méthodologie. Les travaux réalisés pour la construction d'une machine virtuelle défensive, de ses abstractions et du modèle du vérificateur de code octet nous ont permis de fixer et de généraliser la méthodologie appliquée pour assurer la sûreté du typage.

Jakarta. Nous avons participé à la définition des concepts et la création de l'outil *Jakarta*, toutefois, à l'exception de la conception des passerelles vers d'autres langages (voir section 4.4.2), l'essentiel de l'implémentation a été réalisé par Simão MELO DE SOUSA puis Pierre COURTIEU pour la partie liée aux preuves.

CHAPITRE 2

Contexte

Dans ce chapitre, nous présentons le contexte lié à la formalisation de la plate-forme *Java Card*. Nous commençons ainsi par décrire les spécificités et contraintes des cartes à puce. Nous introduisons ensuite l'environnement *Java Card* destiné à la dernière génération des cartes à puce, capable d'exécuter des programmes. Enfin, avant de présenter l'état de l'art dans la modélisation de la plate-forme *Java Card*, nous donnons un aperçu des méthodes formelles et de ses outils.

2.1 Les cartes à puce

La carte à puce vise à stocker et à protéger des informations personnelles. D'une simple capacité de stockage et de contrôle d'accès à ces données personnelles, la carte à puce a peu à peu acquis des capacités de traitement de l'information (telle que les cartes SIM des téléphones cellulaires). Dans ses dernières générations, elle est désormais capable d'exécuter du code et d'héberger divers logiciels, se rapprochant ainsi, à son échelle, d'un ordinateur.

2.1.1 Présentation physique

Une carte à puce est constituée de trois parties :

– un support en plastique, aux dimensions désormais connues sous le

nom de « format carte de crédit », supportant les contraintes d'une utilisation quotidienne ;
- un module de contact, assurant liaison entre la puce et son lecteur ;
- la puce elle-même, située sous le module de contact.

La norme ISO 7816 [67] standardise les caractères physiques de ces cartes, tels que l'emplacement sur la carte du module de contact ou les protocoles de transmission avec la puce.

La puce rassemble en un bloc monolithique (de surface inférieure à 25 millimètres carrés) les composantes courantes d'un ordinateur. On y trouve évidemment un processeur, dont les capacités restent limitées (bus de 8 bits et fréquence de 4,77 MHz pour les modèles usuels) et à évolution nettement moins rapide que le microprocesseur des ordinateurs de bureau. Ces processeurs sont en communication par le bus avec trois types de mémoire :

- la Read Only Memory (ROM), mémoire dont le contenu, non modifiable, inclut le système d'exploitation de la carte et éventuellement certaines applications ;
- la Random Access Memory (RAM), mémoire de travail volatile (son contenu disparaît en l'absence d'alimentation électrique) et à accès rapide ;
- l'Electrically Erasable Read Only Memory (EEPROM) (comme les mémoires Flash) qui s'apparente malgré son nom plutôt à une mémoire de stockage, à contenu non-volatile mais à accès long.

Ici encore, les capacités de ces mémoires sont très restreintes : le rapport entre l'espace mémoire offert par une carte à puce et celui d'un ordinateur personnel est de l'ordre de 1 à un million. On pourra trouver dans [52] un aperçu des fonctionnalités faisant encore défaut, comme le contrôle des ressources et les opérations temps-réel.

Enfin, la carte à puce ne dispose pas d'alimentation électrique propre et dépend pour cela d'une antenne radio pour une utilisation sans contact ou d'un terminal de communication appelé Card Acceptance Device (CAD) . Le lecteur de carte intégré au terminal constitue également l'interface d'entrée/sortie avec la carte.

2.1.2 Utilisation

La carte à puce représente la synthèse actuelle de la dématérialisation de l'argent d'une part (depuis les lettres de changes, titres, chéquiers) et des systèmes d'informations numériques personnels d'autre part.

Dans le domaine des cartes de paiement, Dinner's Club a été le pre-

mier groupe bancaire à proposer dès 1950 une carte évitant l'intervention de monnaie fiduciaire par la donnée des références bancaires de son porteur. Plus tard dotée d'une piste magnétique pour une reconnaissance par une machine, elle verra son évolution principale en terme de sécurité fournie par l'invention de la carte à puce, attribuée à Roland MORENO. Toutefois, les banques ne seront pas les premières à s'intéresser à ce « système d'information » de poche. Les compagnies de téléphone y trouvent un moyen idéal pour le paiement des communications dans les cabines publiques (il s'agit des télécartes en France). Les appareils ne sont plus détériorés pour en retirer la monnaie qu'ils contiennent, les communications sont payées d'avance à l'achat de la carte et constituent ainsi une réserve de trésorerie et enfin, la surface disponible sur la carte permet l'insertion de messages publicitaires. Pour le client, cette avancée lui évite la recherche de monnaie. Le concept sera décliné plus tard pour d'autres types de micro-paiements, par exemple pour les horodateurs.

L'adoption massive de la télécarte conduit à la création du Groupement d'Intérêts Economiques (GIE) Carte Bancaire en vue de l'utilisation de la carte à puce dans le domaine bancaire. La carte bancaire permet à la fois l'identification par son numéro de carte et l'authentification par un code secret, le code Personal Identification Number (PIN) de son porteur. Les transactions sont facilitées, les fraudes réduites : le code PIN est inviolable (il est vérifié par la carte uniquement et confirmé par un système à clés publique et privée), la validité de la carte et la solvabilité du porteur peuvent être vérifiées automatiquement.

Dans le domaine des paiements, les réussites commerciales que constituent la télécarte et la carte bancaire contrastent avec l'adoption limitée du porte-monnaie électronique (carte à puce pour les petits paiements, intégré également à certaines cartes de crédit) sans que l'on sache exactement si cela provient de l'attachement à la monnaie (et de la trop grande dématérialisation que le porte-monnaie électronique génère), du prix des services ou d'une confiance modérée des utilisateurs dans la sécurité du système.

La carte à puce est désormais utilisée dans beaucoup d'autres domaines que le paiement. Ses capacités de stockage d'informations et protection de ces informations sont mises en valeur par des applications telles que la carte de santé, renfermant des données médicales personnelles, ou les cartes de fidélité. Ses facultés d'identification et d'authentification (par le code PIN ou d'autres moyens morphologiques) sont exploités directement pour l'identification de personnes, dans une entreprise pour le contrôle d'accès ou prochainement peut-être dans la vie publique comme carte d'identité.

Enfin, les cartes à puce ont connu ces dernières années un nouvel essor avec la téléphonie mobile. Après l'adoption généralisée de la norme Global

System for Mobile communication (GSM) et du format Subscriber Identification Module (SIM) de la carte à puce, elles sont utilisées à la fois comme élément d'identification et support applicatif. Les cartes SIM permettent alors à l'opérateur de téléphone de proposer des services personnalisés sur le téléphone.

On le voit, la carte à puce offre de plus en plus de fonctionnalités. Devant la multiplication dans la vie courante de ces cartes, il est tentant d'aboutir à des cartes pouvant faire cohabiter plusieurs services. Il s'agit ainsi de cartes multi-applicatives hébergeant divers programmes pouvant éventuellement communiquer entre eux. Mais pour gagner la confiance du public, ces cartes intelligentes (*smart cards*) doivent assurer, avec une architecture beaucoup plus ouverte et complexe, le même niveau de sécurité que leurs prédécesseurs.

2.1.3 Les cartes intelligentes

La dernière génération de cartes à puce apporte la possibilité d'héberger plusieurs application et d'en charger de nouvelles après leur mise en service. Pour cela, elles disposent d'un système d'exploitation évolué et adapté aux impératifs de sécurité que requiert cette nouvelle fonctionnalité. Parmi ceux-ci, citons Smart Card for Windows de Microsoft [94], MultOS du Consortium Maosco [87] (ces deux systèmes sont actuellement tous deux en perte de vitesse), Camille de l'Université de Lille [60], Open Platform de Visa, .NET Card de Axalto (ancienne division Smart Cards de Schlumberger) [108] et bien sûr *Java Card* de Sun Microsystems [121], le plus utilisé.

Ces systèmes présentent de nombreux avantages. Les développeurs évitent l'utilisation d'un assembleur particulier à une puce et peuvent écrire leurs programmes dans des langages de haut niveau, plus faciles à maîtriser. Ils disposent de librairies spécialement développées pour les cartes à puce, comme les fonctions intégrées au système d'exploitation lui-même (pour la plupart il s'agit d'abstractions du support physique, les entrées/sorties par exemple) ou des librairies offrant des fonctions de cryptographie. Le code écrit, traduit en code octet et non plus en assembleur, devient portable sur plusieurs modèles ou versions de puces sans réécriture (pourvu que le système d'exploitation soit le même).

De part la cohabitation sur une même carte de plusieurs programmes, ceux-ci ont la possibilité de s'échanger des données. Les points de fidélité acquis par la location d'une voiture peuvent par exemple être crédités parmi ceux d'une compagnie aérienne partenaire. Le système d'exploitation de la carte doit alors fournir des moyens pour sécuriser ces échanges.

2.2 Java Card

Dans cette section, nous allons présenter la plate-forme *Java Card*, l'une des dernières plates-formes pour carte à puce, et la plus populaire. Conçue par Schlumberger, *Java Card* a été repris en 1996 par Sun Microsystems afin de compléter son offre de développement basée sur la technologie *Java* au domaine de la carte à puce. Autour de nombreux industriels, le Java Card Forum [69], qui est à l'origine des premières spécification de *Java Card*, a pour but de faire connaître, d'exploiter et de développer la plate-forme *Java Card*.

Cette plate-forme est articulée autour du langage *Java* (plus précisément dans sa version actuelle d'un sous-ensemble de *Java*), d'une machine virtuelle appelée Java Card Virtual Machine (JCVM) [121], d'un environnement d'exécution appelé Java Card Runtime Environment (JCRE) [120], d'un ensemble de librairies accessibles par des Application Programmer Interface (API)s [119] et d'un vérificateur de bytecode appelé ByteCode Verifier (BCV). L'articulation de ces différents composants est donné dans la figure 2.1.

La version actuelle du langage *Java Card* est numérotée 2.2, toutefois, nous décrirons dans la suite la version 2.1.1 de cette plate-forme. Les nouveautés apportées par la version 2.2 concernent la mise-à-jour de certaines APIs et la possibilité de retirer des applettes de la cartes, d'appeler des méthodes distantes (Remote Method Invocation), d'ajouter un dispositif de gestion de la mémoire.

2.2.1 Le langage

Le langage *Java Card* peut se résumer à un sous-ensemble de *Java* auquel on aurait retiré les fonctionnalités trop gourmandes en calcul ou en espace mémoire pour être exploitées sur une carte à puce. Il s'agit donc d'un langage à objet, fortement typé, sans pointeurs explicites, à exceptions, à héritage de classe simple et à héritage d'interface multiple. La similitude avec le langage *Java* est si grande que le code source *Java Card* est écrit dans le syntaxe *Java* et compilé par l'intermédiaire d'un compilateur *Java* standard (le code source doit cependant se contenter de n'utiliser que les fonctionnalités supportées par *Java Card*). Le programme compilé porte, comme en *Java*, la dénomination d'*applette* (ou applet) et offre une portabilité comparable aux programmes *Java* de par l'utilisation d'un ensemble de code octet commun.

Par rapport au langage *Java*, et pour des raisons d'occupation mémoire, le langage *Java Card* n'autorise pas les types de données dits larges, tels que

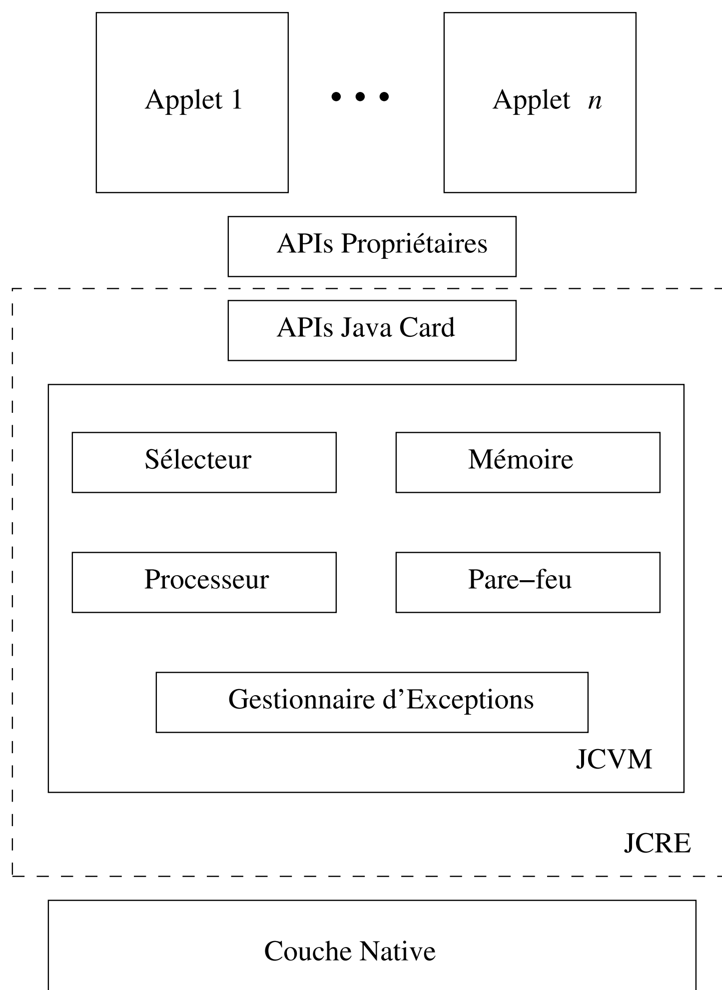

FIG. 2.1 – *Architecture de la plate-forme* Java Card.

les nombres flottants (il n'y a pas en standard d'unité de calcul à virgule flottante sur le microprocesseur des cartes à puce), les chaînes de caractères, les tableaux dont les éléments sont eux-mêmes des tableaux, et la gestion du type des entiers étendus (le type `Int`) est optionnelle.

Parmi les fonctionnalités *Java* non supportées se placent le ramasse-miettes (*garbage collector*, du moins dans la version 2.1.1), les dispositifs de clonage, de finalisation et de sérialisation des classes, le fonctionnement multi-tâches (*multi-threading*). Le chargement dynamique des classes disparaît au profit d'une liaison statique des classes avant leur téléchargement sur la carte, il n'est en effet pas possible de télécharger des classes durant l'exécution d'un programme. Enfin, le gestionnaire de sécurité *Java* est remplacé par un modèle de sécurité spécifique à *Java Card* et lié au vérificateur de code octet.

À titre de comparaison, le code nécessaire à l'implémentation en *Java Card* des chaînes de caractère, des nombres flottants et du gestionnaire multi-tâche prendrait plus de place en mémoire que l'espace ROM disponible.

Le fichier CAP Les programmes *Java Card* ne sont pas représentés comme les programmes *Java* par un fichier *classfile*. Le fichier *classfile* produit par la compilation du code source subit une transformation (dans le sens de la simplification) par le Java Card CAP File Builder afin d'adapter le format de programmes aux ressources minimalistes disponibles sur la carte à puce. Le programme au format Converted APplet (CAP) qui résulte de cette phase de transformation est directement exécutable, a déjà subi l'édition de lien et contient simplement les informations nécessaires à son exécution.

2.2.2 La machine virtuelle

La machine virtuelle *Java Card* est basée sur les notions de piles pour la manipulation des valeurs et de tas pour le stockage des objets. Elle comporte un contexte d'exécution par méthode appelée.

Tout comme la machine virtuelle *Java*, elle exécute du code sous forme de codes octet, c'est-à-dire une représentation par des mnémoniques d'instructions élémentaires, avec éventuellement des opérandes. Ces codes octet seront ensuite traduits dans le langage du microprocesseur de la carte à puce par le système d'exploitation, qui associe à chaque code octet un comportement calculatoire correspondant à sa sémantique. Les codes octet dans le langage *Java Card* sont au nombre de 185.

Une description plus en détails de la machine virtuelle *Java Card* est

donnée, avec sa formalisation, dans le chapitre 3.

2.2.3 L'environnement d'exécution

L'environnement d'exécution *Java Card* peut être comparé au chef d'orchestre de la carte à puce. Il est présent dès le début de vie de la carte, sélectionne les programmes à exécuter sur la carte et est amené à prendre le contrôle de l'exécution par dessus la machine virtuelle, notamment lors d'opérations en rapport avec la sécurité. Il assure également un certain nombre de services aux applettes présentes sur la carte.

La vie d'une applette *Java Card* Après un chargement d'une applette sur la carte à puce, l'environnement d'exécution peut agir sur celle-ci par un ensemble de méthodes devant être implémenté par toute applette. Le JCRE initialise d'abord l'applette après son installation. À partir de ce moment, il peut sélectionner une applette pour lui signifier qu'elle peut s'exécuter, la déselectionner ou lui transmettre un signal APDU (voir plus bas) à interpréter.

Les transactions Une transaction est un mécanisme permettant de rendre un ensemble d'opérations atomique. Par exemple, il est important que lors d'une transaction bancaire les données sur la transaction soient mises à jour entièrement, ou bien pas du tout. Le JCRE offre alors un support aux applettes pour restaurer les données à leur état avant la transaction lorsque celle-ci n'a pas terminé normalement (la carte a été arrachée du lecteur par exemple).

Les objets transients Les applettes requièrent parfois que les données des objets soient temporaires (transientes), c'est-à-dire qu'elles ne persistent pas d'une session d'utilisation de la carte à une autre. Ce type de données est particulièrement adapté aux calculs intermédiaires ou manipulant des données sensibles (telles que des clés cryptographiques) . Bien que *Java Card* ne supporte pas le mot-clé *Java* `transient`, le JCRE permet, par l'appel de méthodes spécifiques, de déclarer certains objets comme transients. Cela interdit alors de les stocker dans la mémoire persistante (EEPROM) et de restaurer leur valeur durant les phases de transaction décrites précédemment.

Les APDUs Le JCRE est aussi responsable des communications avec le terminal CAD. Ces communications se font sous forme d'Application Protocol

Data Unit (APDU)s dont la structure est définie par la norme ISO 7816. Il existe deux catégories d'Application Protocol Data Unit (APDU)s : les APDUs de commande et les APDUs de réponse. Les premiers sont envoyés du lecteur vers la carte et contiennent une commande à exécuter et les derniers sont envoyés en réponse de la carte vers le lecteur. Le JCRE se charge des tampons d'entrée/sortie pour les APDUs, de l'interprétation de ceux-ci et de leur éventuelle répartition aux applettes présentes sur la carte.

Pare-feu Le mécanisme de sécurité de *Java Card* repose sur un modèle de pare-feu (firewall), différent du modèle du bac à sable (sandbox) de *Java*. Chaque applette *Java Card* se voit associer un contexte lors de sa création, à chaque instant, un seul contexte est actif (celui de l'applette courante), et aucune applette ne peut accéder directement à des champs ou des objets d'un contexte différent. Ce cloisonnement disparaît si l'objet accède à des variables statiques, à des tableaux déclarés comme globaux ou bien à des instances de classe déclarées comme points d'entrée (ceci est contrôlé par le JCRE dans les codes octet correspondant à ces fonctions). Dans les autres cas, les applettes de contexte différent peuvent s'échanger des objets implémentant l'interface `Shareable`, dans une procédure toujours contrôlée par le JCRE.

2.2.4 La vérification de bytecode

Parmi les dispositifs de sécurité des plates-formes *Java* ou *Java Card* figure la vérification de code octet. Celle-ci a normalement lieu avant le chargement d'un programme pour la plate-forme *Java* cependant, en raison des capacités limitées des processeurs, elle survient dans le cas de *Java Card* en dehors de la carte à puce, avant la phase de transformation du programme. Des recherches récentes ont toutefois rendu possible une vérification sur la carte même.

Cette vérification consiste d'une part en une vérification structurelle du programme visant à assurer sa bonne formation et d'autre part en une analyse statique du code du programme. Cette analyse est la composante la plus complexe de la vérification car elle doit assurer, entre autres, que les valeurs qui seront manipulées lors de l'exécution auront le type attendu et qu'aucun débordement de la mémoire ne se produira.

Une description plus précise de cette vérification, ainsi que l'état de l'art dans le domaine et sa formalisation, sera donnée dans le chapitre 5.

2.3 Les méthodes formelles

Les méthodes formelles constituent un domaine de recherche très large articulé autour de techniques et d'outils très variés. Fondées sur des bases mathématiques, elles s'appliquent à la modélisation d'un système (ou d'une partie d'un système) et au raisonnement sur ce dernier. Pour cela, les méthodes formelles décrivent :

- les hypothèses sur l'environnement dans lequel le système va évoluer ;
- les contraintes que le système doit établir ;
- une implémentation respectant ces conditions.

Dans le cadre du développement logiciel, elles visent plus spécifiquement à établir que les propriétés posées sur un programme sont bien vérifiées.

Fiabilité par le test

Alors que les logiciels continuent de manière inexorable à croître en fonctionnalités, en complexité et par conséquent en bogues potentiels, les méthodes formelles visent à se substituer à la pratique des tests utilisés alors pour éprouver la fiabilité d'un système. Les tests (ou simulations) consistent à fournir un jeu de données (échantillon) jugé représentatif au système et à comparer les réponses de celui-ci aux résultats attendus. Cependant, l'étendue de l'espace de valeurs à tester (souvent infini) rend impossible pour la majorité des systèmes étudiés le test exhaustif. La difficulté pour éprouver la fiabilité du système se reporte alors sur le choix des valeurs à tester. Et malheureusement, ce choix ne peut que partiellement prendre en compte le fait que les failles d'un système sont dues à des conditions inattendues de son utilisation. Les méthodes formelles se présentent alors, au prix d'une spécification plus rigoureuse, comme la solution la plus sûre pour révéler les incohérences, les ambiguïtés ou l'incomplétude éventuelle des logiciels.

Pour preuve de l'essor des méthodes formelles, l'industrie se met à les adopter, avec succès, par exemple en aéronautique. Même pour les systèmes très complexes, l'utilisation des méthodes formelles en restreignant l'analyse à la partie critique du système apparaît une solution raisonnable en terme de coûts sans pour autant entamer sérieusement la confiance apportée au système.

Nous donnerons dans cette section une vue d'ensemble des méthodes formelles, techniques et outils, toutefois, pour plus de détails, le lecteur pourra se ramener aux références [28, 29, 33, 39, 105, 107]. Nous nous intéresserons alors aux moyens d'évaluer la qualité des modélisations.

2.3.1 Les techniques

L'utilisation des méthodes formelles implique usuellement deux phases de réalisation : la modélisation d'un système et sa vérification.

La modélisation de systèmes

La modélisation s'attache à représenter dans le cadre d'un formalisme mathématique précis un système donné. Les outils actuels et le pouvoir expressif qu'ils ont atteint permettent de modéliser outre des théories mathématiques, d'autres systèmes complexes comme des programmes, des communications entre plusieurs systèmes. Pour mener à bien cette modélisation, ou formalisation, il peut être utile d'abstraire certains détails d'implémentation. On obtient ainsi un modèle plus général, sur lequel il est plus facile de raisonner et d'assurer, en vérifiant certaines contraintes, des propriétés qui seront aussi valables pour le système considéré.

Nous verrons dans la section 2.3.2 qu'il existe un large choix d'outils permettant la modélisation de système. Cependant, les modèles mathématiques sous-jacents à ces outils conditionnent bien souvent un type précis de modélisation (comme celle d'un protocole, d'un programme, d'un algorithme) et les possibilités de raisonnement (automatisme, raffinement, exécutabilité) sur cette modélisation.

La vérification de systèmes

Contrepartie naturelle de la modélisation, la vérification de systèmes déploie un modèle déductif rigoureux pour assurer des propriétés logiques d'un système préalablement modélisé. La vérification se base alors sur un type précis de modélisation, sur un moyen d'exprimer des énoncés logiques sur les systèmes modélisés et sur une procédure de vérification de ces énoncés.

Indécidable sur des modèles aussi simples que le système formel de l'arithmétique de Peano, la démonstration de théorèmes ne peut être automatique sur des systèmes à forte expressivité logique. Seuls certains cas particuliers où la logique est assez restreinte (un fragment de la logique du premier ordre par exemple) ou bien les états du systèmes finis, et donc tous accessibles, permettent un automatisme complet. Les outils proposant cet automatisme sont appelés *vérificateurs de modèle* (ou model checkers).

Dans les autres cas, les outils proposant la vérification de systèmes assisteront leurs utilisateurs dans la construction d'une preuve et vérifieront, de manière complètement automatique cette fois, la preuve fournie. Ces outils sont désignés par les termes d'*assistants à la preuve* ou de *vérifieurs de*

théorèmes.

2.3.2 Les outils

On distingue trois grandes familles d'outils permettant de modéliser des systèmes. Elles se différencient par le modèle mathématique utilisé : algébrique, logique, fonctionnel. Dans cette section, nous présenterons des outils représentatifs de chacune de ces catégories sans toutefois viser à l'exhaustivité. Nous introduirons aussi les outils d'annotation de programmes, plus particulièrement destinés à la vérification de logiciels.

Le modèle algébrique

Nous regroupons dans le modèle algébrique, des outils basés sur des théories mathématiques telles que la théorie des ensembles ou la théorie des catégories (le système Specware [72, 73] par exemple).

Les machines à états abstraits Le modèle des machines à états abstraits ou Abstract State Machine (ASM) [61] est constitué d'une notion d'état et d'ensemble de règles de transitions sur ces états. L'utilisation d'une transition sur un état est conditionné par la vérification de ses conditions de garde et conduit à un nouvel état. Il est possible dans ce formalisme de spécifier un système non-déterministe par la donnée de plusieurs règles de transition sur un même état. Le niveau d'expressivité des ASM se compare aux machines de Türing [123]. Enfin, l'utilisation de règles de transitions permet à ce modèle de fournir très facilement, si les états et les transitions sont décidables, une spécification exécutable. De nombreux outils [66], principalement académiques, sont issus des ASM. Citons par exemple le langage de programmation ASMGofer [109] basé sur les ASM et qui étend le langage de programmation fonctionnelle Gofer (similaire à Haskell).

De manière parallèle aux ASM, l'outil VDM [24], voit le processus de conception dirigé par des raffinements successifs de machines abstraites jusqu'à aboutir à une machine proche de l'implémentation. Enrichie avec la théorie des ensembles de Zermelo-Fraenkel et la logique des prédicats du premier ordre, ce formalisme a donné lieu aux systèmes Z [113] ou B [1], utilisés par les industriels pour d'importantes réalisations (dont la plus connue concerne la ligne de métro METEOR [19]). Dans le système B, chaque étape de raffinement depuis la spécification engendre des *obligations de preuve*. Beaucoup de ces obligations sont démontrées automatiquement mais le reste est à la charge de l'utilisateur qui dispose d'un assistant de preuve intégré.

Les automates Un modèle mathématique proche des ASM est fourni par les automates où l'on retrouve les notions d'état et de transitions sur ces états. Les automates permettent de modéliser des systèmes concurrents, communicants, non-déterministes et fournissent une spécification exécutable. Contrairement aux ASM, les états doivent être connus à l'avance et être en nombre fini. Il s'agit là de la principale limitation des automates, et cela pénalise également la vérification puisque les propriétés sont prouvées par exploration de tous les états. Pour ces raisons, ils sont beaucoup plus adaptés à la vérification de protocoles (pour lesquels le nombre d'états est limité), qu'à la spécification et à la vérification de logiciels.

Le modèle logique

Les modèles logiques centrent la modélisation sur les notion de prédicats, de règles et de réduction sur ces règles. La programmation logique considère les formules logiques comme des programmes et la construction de leur preuve comme l'exécution de ces programmes. Prolog [116] est un des ces outils de programmation logiques. Il s'agit d'un vérificateur de modèle basé sur les clauses de Horn pour la *résolution*, automatique, des preuves.

Un approche assez similaire au modèle logique est donnée par les systèmes de réécriture tels qu'Elan [25] ou Spike [26]. Les axiomes et règles de la logique sont remplacés par un système de règles de réécriture du premier ordre.

Ces systèmes logiques ne sont néanmoins pas adaptés à la formalisation de logiciels car l'écriture de fonctions, même simples, n'y est pas intuitive.

Le modèle fonctionnel

Le modèle de spécification fonctionnel se rapproche des langages de programmation fonctionnels comme Ocaml [82] ou Scheme [53]. Le système modélisé est exprimé par une série de définitions de fonctions, éventuellement récursives, sans effets de bord. Les fonctions sont représentées dans des formalismes dérivés du λ-calcul [7, 38] et sont considérées comme des objets du calcul. Le système de preuves ACL2 [74] utilise le λ-calcul pour langage de spécification combiné à une logique calculatoire pour les preuves. Le λ-calcul peut être enrichi avec des types pour les variables et fonctions [8] et offre alors un modèle de spécification très robuste. Ce λ-calcul typé est la base des systèmes de preuves comme PVS, Isabelle/HOL ou Coq.

PVS [110], développé au SRI, est le plus automatisé (et aussi le plus utilisé) des assistants de preuves. Il repose sur une logique d'ordre supérieur classique et un calcul des séquents pour les démonstrations. Les spécifications

sont dirigées pour les preuves aux dépends de la qualité du code généré.

Isabelle/HOL [98] est un assistant de preuves, développé par les universités de Cambridge et de Munich, basé sur un langage de programmation fonctionnel typé et une logique d'ordre supérieur. Il offre une assez grande automatisation de la construction de preuves et permet d'extraire des spécifications un programme exécutable.

Le langage formel de COQ [89] repose sur le *Calcul des Constructions Inductives* développé à l'INRIA par Thierry COQUAND, Gérard HUET et Christine PAULIN. Il s'agit d'un λ-calcul d'ordre supérieur basé sur l'isomorphisme de Curry-Howard [8]. COQ permet de manier des types dépendants, des constructions inductives et co-inductives et offre ainsi une très forte expressivité. Un théorème est défini par la donnée d'un type et sa preuve par un λ-terme de ce type. Les preuves sont alors objets du calcul lui-même. Pour aider à la construction de ce terme de preuve, COQ dispose de *tactiques* qui sont des pas élémentaires de preuves. L'automatisation reste malheureusement assez faible. En revanche, la logique constructive de COQ facilite l'extraction de programmes à partir des preuves. Enfin, notons que le cœur de COQ, son vérificateur de type, a été prouvé correct de manière formelle dans COQ lui-même [9].

L'annotation de programmes

Pour en finir avec les outils permettant la spécification et la vérification de systèmes, mentionnons ceux proposant l'annotation de programmes. Dans ce cadre, le code source d'un programme est annoté par un énoncé logique décrivant son comportement : les assertions en entrée (les préconditions), les invariants en cours d'exécution et les contraintes en sortie (les postconditions). Il s'agit du modèle de la logique de Hoare. Les annotations sont traduites avec le programme par l'outil vers un système de preuve existant ou dédié pour être vérifiées. Pour le seul langage *Java* et avec le langage d'annotation JML [71], plusieurs outils existent, comme le LOOP Tool [86, 124] construit autour de PVS ou d'Isabelle, Jive [93] et Bandera [6, 64] autour de PVS, JACK [31] autour de l'atelier B, ou encore Krakatoa [88] autour de COQ. Compaq propose l'outil Esc/Java [106] avec un vérificateur intégré et un langage d'annotation spécifique.

L'avantage de cette approche est que l'implémentation elle-même du programme, et non pas une spécification, sert à la vérification. D'autre part, le langage d'annotation est suffisamment simple pour être accessible et utilisé par des personnes non-expertes en méthodes formelles, même si la vérification pourra requérir une expérience plus large.

2.3.3 Une métrique de la modélisation

Avec le développement de la recherche sur le logiciel sûr et les méthodes formelles, il est devenu nécessaire de disposer d'un critère de comparaison, auparavant subjectif, pour juger de la sécurité des systèmes d'information. Entamée à des niveaux nationaux, la recherche d'un modèle d'évaluation de la sécurité a abouti à une approche commune et à la création de la norme ISO-15408, dénommée par *Critères Communs* [46]

Les Critères Communs pour l'évaluation de la sécurité des systèmes d'information définissent des concepts et principes généraux pour l'établissement d'objectifs de sécurité et l'écriture de spécifications haut-niveau.

Dans le processus d'une certification suivant la norme des Critères Communs :

– les développeurs ciblent la partie du système à évaluer et fixent les besoins en terme de sécurité (ils sont aidés pour cela par des *profils de protection* identifiés par les Critères Communs) ;
– les développeurs choisissent une méthode de développement suivant le niveau de sécurité visé ;
– un laboratoire indépendant évalue le niveau d'assurance du développement effectué ;
– une autorité de certification, organisme accrédité, valide le niveau de sécurité suivant le résultat de l'évaluation.

Dans la norme des critères communs, les niveaux d'évaluation, appelés Evaluation Assurance Level (EAL), sont au nombre de 7, posant des contraintes de plus en plus fortes sur le système :

– EAL 1 – Testé de manière fonctionnelle ;
– EAL 2 – Testé de manière structurelle ;
– EAL 3 – Testé de manière méthodique et vérifié ;
– EAL 4 – Conçu de manière méthodique, testé et recensé ;
– EAL 5 – Conçu de manière semi-formelle et testé ;
– EAL 6 – Vérifié de manière semi-formelle et testé ;
– EAL 7 – Vérifié de manière formelle et testé.

L'emploi des méthodes formelles n'est requis qu'au delà du niveau EAL 4. À partir de cette échelle, l'utilisateur est capable de connaître le niveau de sécurité auquel un produit a été évalué et d'acquérir ce produit s'il correspond à ses attentes en terme de sécurité.

De plus en plus de compagnies ont recours à cette évaluation pour faire reconnaître auprès des consommateurs la qualité de leurs produits. Parmi celles-ci, Microsoft a atteint le niveau EAL 4 pour Windows 2000, IBM

a commencé à faire évaluer Linux au niveau EAL 3. Cependant, ces niveaux d'évaluation restent facilement accessibles et envisageables en termes de coût. Les niveaux supérieurs demandent des ressources bien plus importantes. Axalto a vu sa méthodologie de développement pour la construction d'une machine virtuelle *Java Card* conforme aux spécifications de Sun évaluée au niveau maximal EAL 7.

2.4 La sécurité

Dans cette section, nous décrirons la problématique particulière de la sécurité des cartes à puce. Nous décrirons ensuite les travaux liés à cette problématique et principalement les différentes formalisations de la plateforme *Java Card*.

2.4.1 Problématique

En tant que dispositif personnel protégeant ou contenant des données sensibles, la carte à puce se doit d'assurer un niveau sécurité *ad hoc*. Cette sécurité se joue à deux niveaux : la protection de l'accès à la carte elle-même et l'assurance d'un déroulement correct des programmes qui s'y exécutent. McGraw et Felten décrivent dans [90] les risques de sécurité liés à l'utilisation de *Java* et *Java Card*.

Du point de vue de la protection de la carte, c'est-à-dire son utilisation limitée au porteur de la carte, la carte à puce constitue une avancée sensible par rapports aux anciennes cartes magnétiques. Les clés cryptographiques (et le code PIN) ne sont plus lisibles, même sous leur forme encryptée. Ils ne sortent pas de la puce, ni ne sont transmises par des voies de communication. De plus, la puce se bloque d'elle-même après plusieurs essais infructueux du code PIN. Mais la puce rend aussi difficile toute attaque physique :

- son caractère intégré (calculateur, mémoires) et sa surface minuscule limitent les tentatives d'isolation de ses composantes ;
- elle dispose de capteurs détectant son utilisation dans des conditions extrêmes (alimentation électrique, éclairage, signal d'horloge ou température anormaux) qui pourraient révéler des informations sur ce que contient la carte.

La protection de la carte se reporte alors sur la protection du code PIN et sur les logiciels placés sur la puce. Anderson reporte dans [2] les points faibles incontournables que constituent, par négligence, son utilisateur (divulgation du code PIN par exemple) ou, par malveillance, ses intermédiaires

lors de la fabrication (copie d'une carte par exemple). Il mentionne également comment des fonctions crytographiques (comme celles pouvant être contenues sur la carte) mal implémentées peuvent permettre l'attaque de la carte. La certification des logiciels destinés à être placés sur la carte apparaît alors comme une réponse à ce dernier problème.

Les cartes à puce ouvertes ajoutent une nouvelle dimension à la sécurité des logiciels présents sur la carte. Ceux-ci n'y demeurent désormais plus durant toute la vie de la carte. D'autres programmes, issus ou non du constructeur de la carte, peuvent être placés après la mise en service et cohabiter avec les logiciels existants. Le modèle de sécurité reposant uniquement sur la confiance accordée au logiciel du producteur de la carte change alors de manière radicale.

Le problème est d'assurer à l'utilisateur, dans la mesure où il a maintenant la possibilité de charger d'autres programmes sur la carte, que ces programmes n'interféreront pas avec les programmes déjà présents. C'est normalement à la charge des mécanismes de sécurité de la carte (tels que le pare-feu dans *Java Card*) et du vérificateur de code octet d'assurer cette propriété. En conséquence, la sécurité sur la carte repose sur les hypothèses que, d'une part, l'environnement d'exécution et la machine virtuelle ont été correctement implémentés et, d'autre part, que le programme chargé sur la carte a été vérifié.

Le suivi d'un processus de certification, tel que celui apporté par les Critères Communs, conjointement à l'utilisation des méthodes formelles permet d'éviter des failles dans l'environnement d'exécution (cloisonnement des objets, gestion de la mémoire) ou dans la machine virtuelle (interprétation des commandes). La formalisation de ces deux entités, sous une forme exécutable, constitue donc un préalable à la certification. Il s'agit de la technique visant à « éviter les défauts » (fault prevention), la plus sûre pour le but recherché.

Concernant la vérification du programme chargé sur la carte, McGraw et Felten [90, chapitre 2] reconnaissent que la sécurité de *Java* et *Java Card* repose essentiellement sur la sûreté du typage. Pour *Java Card*, celle-ci est normalement assurée par le vérificateur de code octet avant la transformation en fichier *capfile*. Cependant, dans un modèle où les fabricants de carte ne procéderaient pas seuls à chargement de programmes sur la carte, le fichier *capfile* peut très facilement être altéré avant son transfert sur la carte. Il faut alors une garantie au porteur de la carte que le programme est intègre. Celle-ci pourrait lui être fournie par un organisme indépendant, un tiers de confiance. Le porteur d'une carte devrait alors :

– soit se rendre dans un lieu dont il reconnaît l'autorité pour charger un programme sur sa carte, mais cette solution restreint considérablement

la facilité d'extension apportée par les cartes ouvertes ;
– soit n'accepter que des programmes signés par un tiers de confiance avec des protocoles cryptographiques. Dans ce schéma, le programme en provenance du fournisseur d'applettes est transmis à un tiers de confiance. Ce dernier vérifie et signe l'applette. L'applette ne pourra finalement être installée sur une carte qui la signature qui lui correspond est correcte. La sécurité repose ici sur un critère subjectif, la confiance dans le tiers de confiance, et sur un critère objectif, l'implémentation correcte du code cryptographique. Notons cependant que ces fonctions cryptographiques ne garantissent que l'origine d'un programme, pas sa sémantique.

Ces approches, qualifiées de post-issuance, ne sont cependant pas encore appliquées.

On le voit, l'étape de vérification d'un programme avant son chargement sur la carte implique de prendre des précautions complémentaires sur l'origine du programme. La solution la plus efficace en terme de sécurité consiste plutôt à ce que la carte assure entièrement sa sécurité et vérifie elle-même le programme chargé. Cette approche, impensable il y a encore quelques années, peut désormais être réalisée grâce aux techniques de Proof Carrying Code et au concept de vérification «légère» de code octet (voir section 5.1.4).

Notons enfin l'existence d'autres types d'attaques possibles sur une carte, ne compromettant pas sa sécurité, mais pouvant rendre cette carte inutilisable. Parmi celles-ci, l'absence d'un dispositif de ramasse-miettes rend la carte particulièrement sensibles aux programmes malicieux épuisant la mémoire disponible sur la carte (et empêchant ainsi le fonctionnement des autres applettes). Également, la carte peut subir un déni de service par un programme s'emparant cette fois de l'intégralité des ressources de calcul.

2.4.2 État de l'art de modélisation et de la vérification Java (Card)

Dans cette section, nous présentons sur les travaux existants portant sur la modélisation et vérification des plates-formes *Java* ou *Java Card*. Pour une vue d'ensemble encore plus complète de ces travaux, le lecteur pourra se référer à [63]. Notons que nous omettrons ici les travaux, nombreux, portant sur la vérification des programmes *Java* ou *Java* à partir de leur code source, comme avec JML (cf [32]).

ACL2 Le premier effort connu visant à formaliser la machine *Java* a été accompli par COHEN [45] avec le système de preuve ACL2. La formalisation donne une sémantique opérationnelle exécutable et permet d'utiliser, après

conversion, des fichiers *classfile*. Son modèle de machine virtuelle est qualifié de « défensif » car il inclut les vérifications nécessaires pour assurer au cours de l'exécution la sûreté du typage. Sa formalisation, dans un cadre formel qui ne permet pas les ambiguïtés, met à jour certains problèmes d'interprétation de la spécification de Sun. Les instructions de la Java Virtual Machine (JVM) n'ont pas toutes été formalisées (il en manque près de la moitié) et certaines fonctionnalités *Java* (comme les interfaces, la gestion du multi-tâche, du chargement dynamique de classe) ne sont pas traitées. Notons aussi qu'aucune preuve n'a été réalisée sur cette formalisation bien que le système ACL2 le permette.
Les travaux de Cohen sont complétés par Strother Moore et ses étudiants [118] qui y ajoutent la gestion du multi-tâche et développent une preuve formelle de la sûreté du typage.

Projet Bali Mené à l'université de Munich, ce projet [5] vise à formaliser la plate-forme *Java* dans le système de preuve Isabelle/HOL. Les travaux sont menés :

– au niveau du langage source, où le système de types ainsi qu'une sémantique opérationnelle et axiomatique ont été formalisés et où la sûreté du typage a été prouvée [97, 125]. Le langage étudié, appelé Java_{light}, ne couvre cependant qu'une partie du langage *Java* et de ses fonctionnalités (traitement minimal des exceptions, pas de multi-tâche, ...).
– au niveau du code octet, où ont été formalisés une grande part de la JVM ainsi qu'un vérificateur de code octet [78, 100] et un vérificateur de code octet « léger » [77]. La sûreté du typage a également été prouvée et les preuves de correction des vérificateurs de code octet apportées. À nouveau, le langage étudié ne comprend pas l'intégralité des instructions mais plutôt un sous-ensemble représentatif ;
– aux niveaux du langage source et code octet simultanément par la formalisation d'un compilateur [117] de Java_{light} vers un sous-ensemble du code octet de la JVM et la preuve formelle, toujours dans Isabelle/HOL, de sa correction.

Le développement du projet Bali prend ainsi en compte les composants essentiels de la plate-forme *Java*, depuis le langage source jusqu'au code octet de la machine virtuelle. Si le niveau de détail est inégal entre ces différents composants, les vérificateurs de code octet constituent la partie la plus aboutie du projet, avec un traitement de l'initialisation des objets et des sous-routines.

Gemplus Fournisseur mondial de solutions basées sur les cartes à puce, Gemplus a développé une formalisation de la machine virtuelle *Java Card*. Son équipe de recherche a employé pour cela, comme cela devient fréquent dans l'industrie, la méthode B. L'équipe a commencé par spécifier une machine virtuelle défensive [36], c'est-à-dire effectuant les vérifications de type à l'exécution. Cette spécification est ensuite raffinée, selon les techniques de la méthode B, en un vérificateur de code octet et un interpréteur de code octet sans vérification de types (une machine virtuelle offensive). La correction de cette approche tient aux obligations de preuves déchargées lors de chaque étape de raffinement. Ici encore, seulement un sous-ensemble de la JCVM est pris en compte, même si une étude de passage à l'échelle a été effectuée [103]. L'approche, dans la dérivation à partir d'une machine virtuelle défensive d'une vérificateur de code octet est très similaire à nos travaux. Cependant les outils et techniques employées pour arriver au résultat sont fondamentalement différentes. De plus, la forte automatisation des preuves s'effectue aux dépends de l'expressivité de propriétés sur la spécification.
En complément de ces travaux, l'équipe de Gemplus, et plus particulièrement CASSET, a développé un vérificateur de code octet léger, destiné à être embarqué sur une carte à puce [33–35]. Les possibilités d'extraction de l'Atelier B vers du code C ont permis de tester ce BCV sur une carte à puce. Enfin, l'équipe de Gemplus a aussi étudié la conversion d'un sous-ensemble de code JVM vers du code JCVM [81]. Ils obtiennent ainsi une preuve que la spécification de la sémantique du code de la JCVM est un raffinement de données du sous-ensemble de la JVM.

Axalto et Trusted Logic Des chercheurs de Trusted Logic et Axalto ont formalisé un modèle presque complet de la plate-forme *Java Card* [59]. Cette modélisation, effectuée avec l'assistant de preuve COQ, couvre les structures du format *capfile*, tous les codes octet, les APIs basiques et les méthodes natives, un vérificateur de code octet. Elle se trouve actuellement, suivant un long processus, en cours d'évaluation pour un niveau EAL 7. la méthodologie employée a cependant déjà atteint ce niveau.
Le modèle inclut une machine défensive, une machine offensive et deux autres machines abstraites à partir de la machine défensive dont l'une vérifie le typage et l'autre les propriétés du pare-feu. Les preuves de correction de chaque composant vis-à-vis d'une politique de sécurité sont construites. Les machines, en particulier les relations de transition, sont ici exprimées par des prédicats et non par des fonctions. Les preuves sont ainsi facilitées aux dépends d'une exécutabilité plus difficile à obtenir.
Du côté vérification de code octet, LEROY [83] propose un BCV pouvant être embarqué sur une carte à puce. Le BCV requiert une transformation de code et des conditions sur le programme à vérifier, là où les approches actuelles préfèrent les techniques de Proof Carrying Code

Ce travail constitue un effort très important et représente l'équivalent de 6 ans / homme avec 2 à 5 experts en continu sur la formalisation. Les auteurs montrent ainsi que COQ peut être utilisé en milieu industriel, avec un coût certes très élevé mais avec l'accès aux niveaux d'évaluation maximaux. Leurs conclusions vont dans le sens de la construction d'outils permettant de gérer des modèles imposants. Signalons enfin qu'en raison des efforts financiers nécessaires à cette formalisation, cette dernière n'est pas disponible, ce qui nuit extrêmement à la visibilité du projet.

SecSafe Autour du projet SecSafe dédié à l'étude de l'utilisation des techniques d'analyse statique pour les cartes à puce intelligentes et la programmation sur Internet, s'est développée une composante sur la sémantique de *Java Card*. HANKIN et SIVERONI proposent une sémantique opérationnelle à petits pas [62, 112] d'un sous-ensemble de la machine virtuelle *Java Card* dénommé Carmel et comprenant l'essentiel des particularités du langage comme les exceptions, les tableaux, les sous-routines. L'environnement d'exécution et les APIs sont également étudiés [55, 111]. Enfin, l'exploitation de ces travaux dans l'outil PVS est prévue mais non réalisée.

JBook Dans leur livre [114], communément appelé JBook, BÖRGER, SCHMID et STÄRK fournissent une modélisation de l'architecture *Java* basée sur les ASM et le langage de programmation AsmGofer [109]. La spécification du langage *Java* est organisée en une série de cinq couches qui s'étendent successivement, à la manière d'un raffinement : le noyau impératif, la modularité, l'orientation objet, les exceptions et le système multi-tâche. La spécification de la JVM est organisée de la même manière. Pour compléter cette modélisation de l'architecture *Java*, le JBook donne également une spécification d'un compilateur et d'un vérificateur de code octet.
Du côté preuve, le JBook fournit les preuves de sûreté du typage, de correction et de complétude du compilateur. Il est alors possible d'obtenir la propriété principale reliant la vérification d'un programme à son exécution : tout programme *Java* bien formé, bien typé et correctement compilé passe le vérificateur et est exécuté, sans erreur de typage, à l'exécution. De plus son comportement est correct et celui attendu.
Bien qu'il manque certains aspects de l'architecture *Java* comme le ramasse-miette, le chargement dynamique de classe au niveau *Java*, les paquettages, ces travaux constituent un travail considérable et un modèle formel cohérent combinant *Java* et la JVM. Enfin, notons que modèles écrits en AsmGofer sont disponibles et permettent l'exécution.

Autres travaux Parmi les autres travaux qui ne sont pas directement liés à une thématique de recherche globale sur la modélisation des plates-formes

Java ou *Java Card*, les travaux de Jacobs et Poll [68] proposent une sémantique dénotationnelle et axiomatique de *Java* en utilisant monades et coalgèbres. Ce type de sémantique constitue une approche originale pour la gestion des cas d'erreurs de la machine virtuelle.
Sur la formalisation des composantes spécifiques de la plate-forme *Java Card*, mentionnons une modélisation par Éluard et Jensen du pare-feu *Java Card* [54], utilisée pour une analyse de flot vérifiant le contrôle d'accès aux objets. Enfin, Meijer et Poll proposent une spécification des APIs *Java Card* [91], point rarement traité, menée avec le langage de spécification JML. Cette spécification est destinée à être employée avec une formalisation de la machine virtuelle *Java Card* pour une vérification des programmes *Java Card*.

Pour suivre les conclusions de [63], de très nombreux travaux ont été réalisés autour de la modélisation des plates-formes *Java* ou *Java Card*. S'il reste encore à construire un ensemble unifié des modèles formels de *Java* et de ses implémentations afin d'accroître la confiance dans la sécurité de *Java* et d'arriver à une évaluation de niveau EAL 6 ou 7, les travaux effectués constituent un pas essentiel dans cette direction.

2.5 Conclusion

La plate-forme *Java Card* constitue le cadre idéal pour une étude de faisabilité de la formalisation d'un système complexe dans un assistant de preuve comme Coq. En effet, le langage *Java Card* apparaît relativement complet du point de vue des fonctionnalités sans pour autant proposer des particularités trop avancées, comme la gestion du multi-tâche, pour lesquelles les critères d'étude semblent mal définis. Au contraire, les dispositifs critiques du système (les fonctions liées à la sécurité) y sont clairement identifiés.

La formalisation d'un tel système et la vérification de ses parties critiques permettront d'apporter la confiance nécessaire à sa diffusion massive sur les cartes à puce intelligentes. L'assistant de preuve Coq offre pour cela une cadre de développement formel parmi les plus sûrs, permettant d'accéder aux plus hauts niveaux des critères d'évaluation (EAL 7). Les mêmes spécifications seront utilisées pour dérouler les preuves et extraire du code exécutable efficace et alors certifié.

CHAPITRE 3

Formalisation de la machine virtuelle

Nous présentons dans ce chapitre la formalisation en Coq de la machine virtuelle *Java Card* et d'une partie conséquente de son environnement d'exécution. Cette formalisation a été réalisée en gardant constamment en tête les critères suivants :

Passage à l'échelle. La plupart des spécifications formelles de la plate-forme *Java Card* se concentrent seulement sur un sous-ensemble du langage. Il s'agit clairement d'un problème car des brèches de sécurité peuvent survenir d'interactions non prévues entre différentes fonctionnalités. De plus, les méthodologies proposées pour raisonner sur sous-ensemble formalisé peuvent ne plus s'appliquer à l'intégralité de la plate-forme en raison d'une trop grande complexité ou d'un comportement spécifique. Au contraire, notre formalisation prend en compte l'intégralité des instructions de la machine virtuelle *Java Card* et l'essentiel de l'environnement d'exécution. De surcroît, nous proposons des outils (voir chapitres 4 et 6) pour raisonner sur une formalisation de cette taille.

Réalisme et précision. En basant notre formalisation sur les spécifications de Sun [120], nous assurons une compatibilité pour l'exécution des programmes *Java Card* vis-à-vis des autres machines virtuelles de référence, comme celle de Sun. De plus, grâce au caractère exécutable de notre formalisation, nous pouvons comparer les résultats d'exécution de programmes à ceux des machines virtuelles de référence.

Adéquation à la preuve. La formalisation de la machine virtuelle servant de support au raisonnement, il apparaît important que celle-ci soit écrite

de manière à faciliter les preuves. Néanmoins, s'il est possible d'optimiser l'écriture de la formalisation pour rendre plus aisée certaines preuves, la meilleure optimisation pour le cas général consiste à disposer d'une formalisation utilisant un style le plus simple et neutre possible (pas de fonctions d'ordre supérieur, pas de types dépendants, ...). On se rapproche ainsi au maximum d'une spécification donnée par un langage de programmation classique, séparant ainsi les compétences et permettant aussi probablement de proposer des outils de preuves adaptés à d'autres spécifications.

La machine virtuelle présentée ci-dessous (et dans [14, 15]) constituant la base de notre développement pour la vérification de code octet (voir chapitre 5), ces points nous apparaissent cruciaux pour disposer d'une plateforme réaliste et fiable.

3.1 Notations de COQ

Dans la suite de ce chapitre, nous allons être amenés à décrire la structure de la machine virtuelle *Java Card*. Il s'agit principalement de considérations algébriques et il aurait donc été possible d'utiliser des notations mathématiques. Cependant, la machine virtuelle ayant été formalisée en COQ [22, 89], nous avons choisi d'utiliser les notations de ce système, parfois légèrement différentes, que nous allons introduire dans cette section.

On utilise `*`*@`*` pour désigner le produit cartésien de deux types et `(a,b)` pour écrire une paire.

En l'absence de caractère ASCII pour λ et $\forall$, la λ-abstraction $\lambda x : A.t$ s'écrit en COQ avec `[x:A] t` et le produit dépendant $\forall x : A.B$ s'écrit `(x:A) B`. Le produit non-dépendant se note lui de manière classique `A→B`.

Un type inductif est déclaré avec le mot-clé **`Inductive`**, suivi de son nom, éventuellement de paramètres, de son type et d'une liste de constructeurs avec leur nom et leur type. Par exemple, le type des entiers naturels `nat` est défini en COQ par :

```
Inductive nat : Set :=
  O : nat
| S : nat→nat.
```

Un type d'enregistrement est déclaré quant à lui avec le mot-clé **`Record`** suivi de son nom, de son type et d'une description (nom et type) de ses champs. Il est représenté de manière interne par un type inductif à un seul constructeur dont les arguments sont les champs de l'enregistrement. On accède ensuite aux champs de l'enregistrement par des fonctions sélecteurs

et on écrit alors `(f r)` plutôt que le standard `r.f` pour accéder au champ `f` de l'enregistrement `r`.

Les définitions sont introduites avec le mot-clé **`Definition`** suivi du nom donné à la définition, du type de l'objet défini (optionnel) et du corps de la définition.

Le filtrage de constructeur utilise le mot-clé **`Cases`** suivi de la variable filtrée, dont le type doit être un type inductif, et du résultat pour chacun des constructeurs du type. Éventuellement, on peut utiliser le caractère _ (wildcard) pour donner un résultat par défaut aux constructeurs qui n'auront pas été énumérés.

L'exemple suivant illustre une définition de fonction effectuant un filtrage sur le type des entiers naturels `nat` :

```
Definition is_zero : bool := [n:nat]
Cases n of
  O ⇒ true
| (S p) ⇒ false
end.
```

On notera que toutes les fonctions en Coq doivent terminer (dans le cas des points fixes) et être totales. Pour gérer les fonctions partielles, on utilise le type `option` des librairies de Coq (qui fournit une valeur par défaut, `None`, dans les cas de partialité) :

```
Inductive option [A : Set]  : Set :=
  Some : A→(option A)
| None : (option A).
```

Par exemple, la fonction qui retourne la tête d'une liste est définie en utilisant le type `option` pour fournir une valeur par défaut à la tête d'une liste vide :

```
Definition head [A:Set,l:list A] : (option A) :=
Cases l of
  nil        ⇒ (None A)
| (cons x _) ⇒ (Some A x)
end.
```

3.2 Représentation des programmes

Du point de vue de la machine virtuelle, un programme est une représentation structurée du code source écrit par le programmeur. Pour *Java Card*, ce code source a suivi diverses phases, dont les plus importantes sont la compilation et la conversion (voir section 3.6.1). À l'issue de ces phases restent les composantes essentielles du langage à object qu'est *Java Card* : les types, les méthodes, les classes et les interfaces.

La figure 3.1 donne une vue synthétique de la représentation d'un programme.

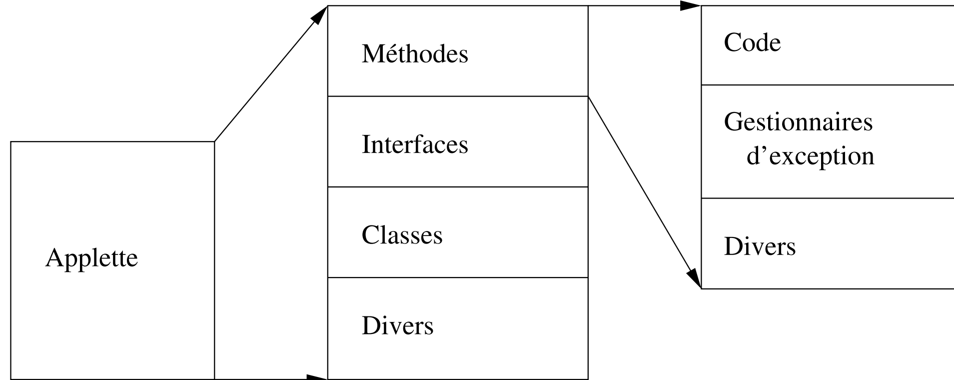

FIG. 3.1 – *Organisation des applettes.*

3.2.1 Le système de types

Comme la plupart des langages de programmation moderne, *Java Card* dispose d'un système de types afin d'assurer une utilisation correcte des valeurs. *Java Card* différencie, comme *Java*, les types primitifs des types référence.

Les types primitifs englobent les types arithmétiques (`Byte`, `Short` et `Int`, qui se distinguent par la plage de valeurs possibles), le type des valeurs booléennes (`Boolean`), le type des adresses de retour des sous-routines (`ReturnAddress`), et enfin le type `Void` des fonctions sans valeur de retour. Ces types se traduisent en COQ avec l'utilisation d'un type inductif :

```
Inductive type_prim : Set :=
  Byte          : type_prim |
  Short         : type_prim |
  Int           : type_prim |
  Boolean       : type_prim |
  ReturnAddress : type_prim |
  Void          : type_prim.
```

Les types référence regroupent quant à eux le type des instances de classe, des interfaces, des tableaux et des références nulles. Notons que l'on précise pour le type des instances de classe la classe à laquelle on fait référence (de même pour les interfaces) et pour le type des tableaux le type des éléments du tableau. Le type des éléments du tableau étant lui-même un type *Java Card* (primitif ou référence), on est conduit à définir simultanément les types *Java Card* et les types référence. On utilise alors la construction de types mutuellement inductifs de COQ :

```
Mutual Inductive type : Set :=
  Prim          : type_prim → type |
  Ref           : type_ref → type
with type_ref : Set :=
  Ref_null      : type_ref |
  Ref_array     : type → type_ref |
  Ref_instance  : class_idx → type_ref |
  Ref_interface : interf_idx → type_ref.
```

Cette définition des types, en particulier des types de tableaux, permet de contruire le type des tableaux dont les éléments sont eux-mêmes des tableaux alors que les tableaux de tableaux ne sont pas permis dans *Java Card* (ils le sont dans *Java*). Néanmoins, il est facile de montrer (par une analyse statique du programme et de l'instruction de création de tableaux `anewarray`) qu'un tel type ne sera pas utilisé.

3.2.2 Les méthodes

Une méthode est le nom donné aux fonctions définies à l'intérieur d'une classe (voir 3.2.4). Une méthode est caractérisée par son statut (statique ou non, constructeur ou non), sa signature (pour la vérification du typage de ses arguments lors d'un appel de méthode), son nombre de variables locales (pour l'initialisation de son contexte d'exécution), son code à exécuter (le code octet, sous forme de liste d'instructions), ses gestionnaires d'exception (pour la gestion dynamique des erreurs), sa taille maximum de pile d'opérande lors de son exécution (pour contrôler les débordements de pile) et enfin la classe à laquelle elle appartient. On obtient alors l'enregistrement suivant :

```
Record Method : Set := {
  (* static method or not *)
  statusstatic  : bool;

  (* init method or not *)
  statusinit    : bool;

  (* signature, pair of domain / codomain *)
  signature     : ((list type)*type);

  (* number of local variables *)
  local         : nat;

  (* instructions to be executed *)
  bytecode      : (list Instruction);

  (* exception handlers *)
  handler_list  : (list handler_type);

  (* maximum operand stack size *)
```

```
  m_max_opstack_size : class_idx;

  (* owner class index *)
  owner          : class_idx;

  (* method identification *)
  method_id      : method_idx
}.
```

où `class_idx` correspond au type des index vers des classes dans le fichier *capfile* et `method_idx` à celui des méthodes (ces index sont des entiers naturels). Le type `handler_type` est quant à lui décrit dans la section 3.5.3 sur le traitement des exceptions.

Enfin, le type `Instruction` énumère sous forme d'un type inductif, les différents codes octet du langage *Java Card* avec leurs opérandes :

```
Inductive Instruction : Set :=
  nop             : Instruction |
  push            : type_prim → Z → Instruction |
  ret             : locvars_idx → Instruction |
  invokespecial   : nat → method_idx → Instruction |
  invokestatic    : nat → method_idx → Instruction |
  getfield        : type → instance_field_idx → Instruction |
  inc             : type_prim → Z → nat → Instruction |
...
```

Certains codes octet du langages *Java Card* à la sémantique similaire ont été factorisés. Par exemple, le code octet `inc` prend ici comme opérande supplémentaire un type primitif et sert ainsi à représenter les codes octet `iinc` et `sinc` de *Java Card* (pour l'incrémentation d'une variable locale de type `Int` et `Short` respectivement). De même, des codes octet comme `getfield` existent normalement sous les formes `getfield` et `getfield_w` qui correspondent à des tailles différentes d'index (`_w` pour wide) dans la zone des constantes, mais la résolution de celui-ci par le *JCVM Tools* (voir section 3.6.2) permet de n'en conserver qu'un seul.
On peut ainsi représenter les 185 codes octet *Java Card* par seulement 44 codes octet dans notre formalisation.

3.2.3 Les interfaces

Les interfaces définissent un ensemble de signatures nommées de méthodes et de valeurs constantes. Elles servent principalement au développement de librairies. Une classe peut implémenter une interface (éventuellement plusieurs) si elle définit un corps, sous forme de méthode, à chacune des signatures de l'interface.

La plupart des informations sur les interfaces sont inutiles à notre usage.

En effet, le vérificateur de liens assure qu'une classe implémente correctement une interface. Les renseignements sur les méthodes implémentées sont intégrés directement dans le descripteur de classe. La seule information nécessaire sur les interfaces est la relation d'héritage entre ces dernières, pour vérifier par exemple la compatibilité entre deux tableaux d'interfaces. On utilise la structure suivante pour représenter les interfaces :

```
Record Interface : Set := {
  (* super interfaces *)
  super_int : (list interf_idx);

  (* interface identification *)
  int_id    : interf_idx
}.
```

où `interf_idx` correspond au type des index vers des interfaces du fichier *capfile*. Ces index sont des entiers naturels.

Parmi les interfaces *Java Card*, l'interface `Shareable`, joue un rôle particulier dans le système de sécurité *Java Card* (voir section 2.2.3).

3.2.4 Les classes

Les classes décrivent dans les langages à objet les caractéristiques d'objets par un ensemble de données et le moyen, au travers de méthodes, de les manipuler. De plus, les classes traduisent avec les notions d'héritage et d'interface les relations entre objets.

Dans notre représentation, une classe est naturellement décrite par le type de ses variables, par l'ensemble des classes dont elle hérite – ses superclasses –, par les méthodes qu'elle contient (dont ses constructeurs), par les interfaces qu'elle implémente et par le paquettage (voir ci-dessous) auquel elle appartient. Enfin, on garde trace de l'identifiant de la classe dans notre représentation finale du programme. Formellement, en Coq, cela conduit à la représentation suivante, sous forme d'enregistrement :

```
Record Class : Set := {
  (* super classes *)
  super           : (list class_idx);

  (* public methods *)
  public_methods  : (list method_idx);

  (* package methods *)
  package_methods : (list method_idx);

  (* implemented interfaces and corresponding implemented methods *)
  int_methods     : (list interf_idx*(list class_pbmethod_idx));
```

```
  (* list of class variables *)
  class_var       : (list type);

  (* owner package *)
  package         : Package;

  (* class identification *)
  class_id        : class_idx
}.
```

où `class_pbmethod_idx` correspond au type des index du contenu du champ `public_methods` (les méthodes implémentant une interface sont déclarées publiques).

Notons que le champ `super`, qui est une liste, rassemble non pas la classe dont la classe considérée hérite directement (puisqu'en *Java* il n'y a pas d'héritage multiple) mais la fermeture transitive de la relation d'héritage. Par convention, la super-classe immédiate se trouve en première position dans la liste et celle-ci existe toujours : il s'agit par défaut de `java.lang.Object`. Cette représentation a été choisie pour éviter d'avoir à définir des fonctions par récurrence bien-fondée sur la relation de sous-typage.

Notons aussi que l'on distingue parmi les méthodes celles marquées avec les attributs `public` ou `protected` de celles marquées avec les attributs `package` ou `private` et que cette distinction est déjà présente dans le fichier *capfile*.

Enfin, le type `Package` est un identifiant unique de paquettage, c'est-à-dire dans la terminologie *Java* un ensemble de classes et d'interfaces formant un programme. Puisque plusieurs paquettages peuvent cohabiter sur la même carte à puce, l'identifiant est utile pour le contrôle d'accès aux classes, champs statiques et méthodes statiques. Pour des raisons de sécurité, chaque classe appartient donc à un seul paquettage. L'identifiant de paquettage est appelé Applet IDentifier (AID).

3.2.5 Les programmes

Notre modèle de programmes ne possède pas de zone des constantes, contrairement à la représentation d'un fichier *capfile*. On suppose les phases de liaison déjà effectuées (par le *JCVM Tools*, voir section 3.6.2). Ainsi, la définition d'un programme *Java Card* pour notre formalisation se limite quasi exclusivement à une collection de classes, d'interfaces et de méthodes :

```
Record jcprogram : Set := {
  classes    : (list Class);
  methods    : (list Method);
  interfaces : (list Interface);
  sheap_type : (list type)
```

`}.`

On remarquera simplement l'ajout le champ `sheap_type` qui détaille les types des valeurs statiques du programmes et qui servira lors de l'initialisation du programme.

Dans cette représentation de programme, toutes les classes nécessaires à l'exécution sont incluses (il n'y a pas de chargement dynamique de classes en *Java Card*). En particulier, on y retrouve par défaut les interfaces et classes du paquettage `java.lang`.

3.3 Représentation de l'environnement d'exécution

La machine virtuelle *Java Card* fonctionne comme une machine à états et est décrite par une sémantique à petits-pas. Plus précisément, chaque instruction agit comme un transformateur d'état, c'est-à-dire prend en entrée un état et retourne un nouvel état à l'issue de son exécution.

Dans notre formalisation, l'état contient l'intégralité des éléments manipulés par un programme *Java Card* durant l'exécution : valeurs, tas et contextes d'exécution de méthode.

La figure 3.2 donne une vue synthétique de l'organisation de l'environnement d'exécution.

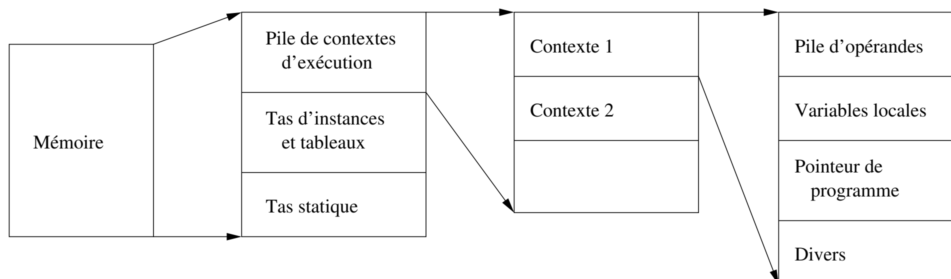

FIG. 3.2 – *Organisation de l'environnement d'exécution.*

3.3.1 Les valeurs

Les valeurs constituent les principaux éléments manipulés par la machine virtuelle. Les valeurs sont typées et suivent donc une définition similaire au système de types (cf section 3.2.1). On retrouve donc les valeurs primitives et les valeurs références.

L'information de type est transportée en même temps que la valeur par

l'utilisation d'un type inductif dont le constructeur signale le type et l'argument contient la valeur. Cela donne :

```
Inductive valu_prim : Set :=
  vByte          : Z → valu_prim |
  vShort         : Z → valu_prim |
  vInt           : Z → valu_prim |
  vBoolean       : Z → valu_prim |
  vReturnAddress : bytecode_idx → valu_prim.
```

On peut ainsi choisir de représenter différemment en Coq certaines valeurs (les valeurs arithmétiques appartiennent à `Z` et les valeurs des types de retour à `bytecode_idx` qui est un entier naturel). On remarquera que, par rapport au système de types, il n'y a pas de valeur associée au type `Void` (c'est la signification de ce type). Enfin, nous fournissons également un autre type d'entiers binaires, `Z_bits`, pour remplacer le type `Z` et gérer alors les débordements de capacité suivant les spécifications de Sun (le type `int` est représenté sur 16 bits et `short` sur 8 bits dans la machine virtuelle *Java Card*).

Les valeurs des références sont définies comme suit :

```
Inductive valu_ref : Set :=
  vRef_null      : valu_ref |
  vRef_array     : type → heap_idx → valu_ref |
  vRef_instance  : class_idx → heap_idx → valu_ref.
```

où `heap_idx` désigne l'emplacement de l'objet en mémoire (il n'est donc pas utilisé pour `vRef_null`, valeur générique des objets).
On remarquera également qu'il n'y a pas de valeurs pour les interfaces, une valeur vers une instance de classe pouvant être considérée dynamiquement comme une interface si l'instance l'implémente.

Enfin, on utilise les définitions précédentes pour construire le type Coq des valeurs *Java Card* :

```
Inductive valu : Set :=
  vPrim    : valu_prim → valu |
  vRef     : valu_ref → valu |
  vNonInit : class_idx → bytecode_idx → heap_idx → valu.
```

`vNonInit` est une valeur spécifique des instances de classes non encore initialisées. Ce point sera traité dans la section 5.1.5.

3.3.2 Les objets

Les objets en mémoire constituent la représentation dynamique des classes et des tableaux. Les valeurs `vRef_instance` et `vRef_array` définies à la section précédente font référence à ces objets.

On regroupe à nouveau dans un enregistrement pour chacun de ces types d'objets les informations nécessaires à l'exécution.

Pour une instance de classe, on précise la classe instanciée, le contenu des variables de classe, le paquettage propriétaire (pour les mécanismes de sécurité) et des informations précisant si l'instance est déclarée comme un point d'entrée (accessible de n'importe quel paquettage) :

```
Record type_instance : Set := {
  reference       : cap_class_idx;
  contents_i      : (list valu);
  owner_i         : Package;
  ptE             : status_ptE
}.
```

où `status_ptE` est un type inductif indiquant s'il s'agit d'un point d'entrée et le cas échéant son statut (temporaire ou permanent).

Pour un tableau, on précise le type statique utilisé lors de la création du tableau, le contenu du tableau, le paquettage propriétaire et un booléen indiquant si le tableau a été déclaré global (accessible de n'importe quel paquettage, concept similaire au point d'entrée des instances) :

```
Record type_array : Set := {
  type_contents : type;
  contents_a    : (list valu);
  owner_a       : Package;
  global        : bool
}.
```

Afin de déclarer le type générique des objets, on utilise le type inductif suivant qui regroupe les définitions `type_instance` et `type_array` :

```
Inductive obj : Set :=
  Instance : type_instance → obj
| Array    : type_array → obj.
```

Enfin, la zone mémoire, appelée *tas*, dans l'état de la machine virtuelle qui regroupe tous les objets créés lors de l'exécution d'un programme est naturellement définie comme une liste de `obj` :

```
Definition heap := (list obj).
```

3.3.3 Les contextes d'exécutions de méthode

Lors de l'appel d'une méthode un contexte d'exécution de méthode est créé et empilé aux contextes déjà existants. Ces contextes renferment les informations calculatoires propres à chaque méthode : la pile d'opérande et les variables locales (aussi appelées registres). Un index indique la méthode

à laquelle se réfère ce contexte et un pointeur de programme, la prochaine instruction à exécuter à l'intérieur de cette méthode. À nouveau, le paquettage auquel appartient le contexte est précisé. Enfin, pour éviter d'avoir fréquemment à rechercher dans la méthode exécutée, après déférencement, la taille maximale de la pile d'opérande, cette information est dupliquée dans le contexte.

```
Record frame : Set := {
  (* operand stack *)
  opstack       : (list valu);

  (* local variables *)
  locvars       : (list (option valu));

  (* location of the method *)
  method_loc    : cap_method_idx;

  (* program counter *)
  p_count       : bytecode_idx;

  (* context information *)
  context_ref   : Package;

  (* max opstack size *)
  max_opstack_size : nat
}.
```

Le type `option` du champ `locvars` servira à traduire le caractère non initialisé d'une variable locale.

La pile de contexte est alors définie simplement sous forme de liste. C'est sur la tête de cette liste (la méthode en cours d'exécution) qu'agiront la plupart des instructions :

```
Definition stack := (list frame).
```

3.3.4 Les états

La plupart des définitions nécessaires pour définir les états de la machine a déjà été introduite (tas, pile de contextes d'exécution, et définitions connexes). Rajoutons seulement le tas statique qui contiendra les valeurs des variables déclarées comme statiques (et donc partagées par tout le programme) :

```
Definition sheap := (list valu).
```

et il est maintenant possible de préciser la notion d'état :

```
Record jcvm_state : Set := {
  sheap_f : sheap;
```

```
  heap_f  : heap;
  stack_f : stack
}.
```

L'état est exprimé par un enregistrement pour plus de souplesse d'utilisation par rapport à un triplet (en COQ, une paire dont une composante est à nouveau une paire). Cela permet également de rajouter des champs (comme les tampons d'entrée/sortie des APDUs, voir section 3.6.6) sans changer le code de chacune des instructions.

Cette notion d'état servira enfin à la construction du type des états de retour des instructions, sachant qu'une instruction peut se dérouler normalement, provoquer une erreur ou lancer une exception :

```
Inductive returned_state: Set :=
Normal         : jcvm_state → returned_state |
Abnormal       : eLabel → jcvm_state → returned_state |
ThrowException : xLabel → jcvm_state → returned_state.
```

Le type `eLabel` est un type inductif dont les constructeurs décrivent le type d'erreur (voir section 3.5.1) ayant provoqué la terminaison du programme (erreur du programme, de typage, de pile vide, exception non récupérée, ...) et `xLabel` précise l'exception lancée (pointeur nul, exception de sécurité, ...).

3.4 Représentation des instructions

Les instructions constituent, par leur enchaînement, le cœur des programmes. Dans la machine virtuelle *Java Card*, elles peuvent être classées en huit catégories principales :

– opérations arithmétiques (`sadd`, `idiv`, `sshr`, ...);
– vérifications de type sur les objets (`instanceof`, ...);
– branchements, éventuellement conditionnels (`ifcmp`, `goto`, ...);
– appels de méthode (`invokestatic`, `invokevirtual`, ...);
– opérations sur les variables locales (`iload`, `sstore`, ...);
– opérations sur les objets (`getfield`, `newarray`, ...);
– opérations sur la pile d'opérande (`ipush`, `pop`...);
– retours de méthode (`sreturn`, `throw`, ...).

Dans notre formalisation, la sémantique de chacune des instructions est donnée par une fonction de type :

```
jcvm_state * operands → returned_state
```

où `operands` est variable d'une instruction à l'autre et correspond aux opérandes attendus par l'instruction. Il s'agit d'une sémantique opérationnelle

à petit pas.

La plupart des instructions ont un motif similaire d'exécution :

1. l'état initial est décomposé ;
2. des composants de cet état sont récupérés ;
3. des tests sont réalisés pour détecter d'éventuelles erreurs ;
4. l'état final est construit sur la base des composants récupérés et des tests effectués.

Dans notre description, la première étape est réalisée en Coq avec l'utilisation de filtrage (**Cases**) ; la deuxième utilise le plus souvent des accesseurs sur les listes ; l'étape suivante permet de déterminer si le nouvel état est un état d'erreur ; enfin la dernière étape correspond à la mise à jour de la mémoire.

Dans cette section, nous détaillons la sémantique d'instructions représentatives : `new` qui créé une nouvelle instance de classe en mémoire ; `if`, l'instruction de branchement conditionnel ; `getfield` pour accéder au champ d'une instance de classe et `invokevirtual` qui appelle une méthode virtuelle.

3.4.1 Sémantique de new

L'instruction `new` prend comme opérande un index de classe. La fonction Coq NEW donnant la sémantique de cette instruction reçoit en plus, comme toutes les fonctions traduisant la sémantique d'une instruction dans notre formalisation, l'état courant de la machine virtuelle et le programme exécuté.

```
Definition NEW := [idx:class_idx][state:jcvm_state][cap:jcprogram]
Cases (stack_f state) of
nil ⇒ (AbortCode state_error state) |
(cons h lf) ⇒
  (* Extract the owner class from thew cap_file *)
  Cases (Nth_elt (classes cap) idx) of
  None ⇒ (AbortCode class_membership_error state) |
  (* then a new instance is created and pushed into the heap *)
  (Some cl) ⇒
    let new_obj = (Build_type_instance idx
                          (init_var (class_var cl))
                          (context_ref h)
                          false
                          false) in
    let nhp = (app (heap_f state) (cons new_obj (nil obj))) in
      (update_frame
        (update_opstack
          (cons (vNonInit idx (p_count h) (S (length (heap_f state))))
                (opstack h)) h)
        (Build_jcvm_state (sheap_f state) nhp (stack_f state)))
```

```
end end.
```

L'état est décomposé pour obtenir le contexte courant d'exécution de méthode. Puis, on accède dans le programme à la classe précisée par l'opérande et on crée une nouvelle instance, de type `obj`, de cette classe que l'on concatène au tas. L'état final est alors obtenu en utilisant le nouveau tas et en ajoutant une valeur de référence à l'objet créé (instance non initialisée avec indication de la classe, du pointeur du programme lors de la création de l'instance et l'emplacement dans le tas) dans la pile d'opérande. Ces opérations sont effectuées par `update_opstack` qui remplace dans un état la pile d'opérande par celle fournie en argument (et augmente le pointeur de programme) et par `update_frame` qui met à jour le contexte d'exécution de méthode courant dans l'état donné en argument.

Dans tous les cas d'erreur, un état anormal est construit avec la fonction `AbortCode` et une étiquette expliquant la raison de l'erreur.

Par construction, la fonction `NEW` ne peut pas rendre une référence nulle (`Ref_null` ou `vNonInit` avec le dernier argument du constructeur, ici `(S (length (heap_f state)))`, égal à zéro). Il est cependant possible de modifier la fonction `NEW` pour qu'elle simule également le comportement d'une machine virtuelle dans le cas d'un manque de mémoire en ne créant pas l'objet et en rendant une référence nulle.

3.4.2 Sémantique de if_acmp

L'instruction `if_acmp` effectue un branchement suivant la comparaison de deux valeurs de références. Ces valeurs sont dépilées de la pile d'opérande et doivent être de type `Ref`.

L'instruction `if_acmp` existe normalement dans le jeu d'instructions de la JCVM dans quatre formes. Les formes `if_acmpeq` et `if_acmpne` testent respectivement pour le cas de réussite si les valeurs sont égales ou si elles sont différentes. Ces deux formes sont rassemblées en une fonction Coq unique en utilisant une opérande supplémentaire pour déterminer la comparaison à effectuer. Il existe aussi les formes `if_acmpeq_w` et `if_acmpne_w` où le suffixe `_w` (pour *wide index*) indique d'utiliser une adresse de branchement sur deux octets. Cependant, il n'est pas utile de se préoccuper de cette différenciation puisque le *JCVM Tools* a déjà converti ces index en entiers naturels. Ainsi, et pour toutes les instructions utilisant ce type d'index, la sémantique de `if_acmpeq_w` et `if_acmpne_w` est donnée par la sémantique de l'instruction correspondante standard.

Pour ces instructions de branchement, l'état est décomposé pour en sortir les deux valeurs du haut de la pile d'opérande. On vérifie ensuite qu'il s'agit

bien de valeurs de référence et on teste si ces valeurs sont égales ou non. En fonction du résultat du test, le pointeur de programme est mis à jour.

La fonction COQ `IF_ACMP_COND` est la fonction appelée par le répartiteur d'instructions pour appliquer la sémantique des codes octet `if_acmp`.

```
Definition IF_ACMP_COND :=
[oper:opcmp][branch:bytecode_idx][state:jcvm_state]
Cases (stack_f state) of
nil ⇒ (AbortCode state_error state) |
(cons h lf) ⇒
  (* Extract the 2 values a the top of the opstack *)
  Cases (head (opstack h)) of
  None ⇒ (AbortCode opstack_error state) |
  (Some v2) ⇒
    Cases (head (tail (opstack h))) of
    None ⇒ (AbortCode opstack_error state) |
    (Some v1) ⇒
      (* Test for references *)
      Cases v1 v2 of
      (vRef vx) (vRef vy) ⇒
        (update_frame (res_pc2
          (res_acompare2 oper (vr2hi vx) (vr2hi vy))
           branch h)
         state) |
      _         _         ⇒ (AbortCode type_error state)
end end end end.
```

La fonction `res_pc2` pour mettre à jour le contexte d'exécution courant (résultat du branchement pour le pointeur de programme et pile d'opérande où les deux valeurs ont été otées) suivant le résultat de la fonction de comparaison `res_acompare2`.

```
Definition res_acompare2 := [oper:opcmp][z1,z2:Z]
Cases oper of
  Zeqb ⇒ (Zeq_bool z1 z2)  |
  Zneb ⇒ (Zneq_bool z1 z2) |
  _    ⇒ false
end.

Definition res_pc2 := [result:bool][branch:bytecode_idx][h:frame]
(if result
then (update_pc branch
       (update_opstack (tail (tail (opstack h))) h))
else (update_pc (S (p_count h))
       (update_opstack (tail (tail (opstack h))) h))
).
```

3.4.3 Sémantique de getfield

L'instruction `getfield` accède au contenu d'un champ d'une instance de classe. La référence à l'objet est dépilée de la pile d'opérande ; il doit s'agir d'une référence non nulle (sans quoi une exception dynamique `NullPointerException` est levée, cf section 3.5.2) à une instance de classe initialisée. Cette référence est utilisée pour accéder à l'objet dans le tas. La valeur contenue à l'index demandé (une des opérandes de l'instruction) parmi les champs de l'instance est extraite et poussée dans la pile d'opérande. Enfin, le type de la valeur extraite doit correspondre au type spécifié par l'opérande de l'instruction.

```
Definition GETFIELD := [t:type][idx:instance_field_idx]
[state:jcvm_state][cap:jcprogram]
Cases (stack_f state) of
nil ⇒ (AbortCode state_error state) |
(cons h lf) ⇒
  Cases (head (opstack h)) of
  None     ⇒ (AbortCode opstack_error state) |
  (Some x) ⇒
    Cases x of
    (vNonInit _ _ _) ⇒ (AbortCode init_error state) |
    (vRef vr) ⇒
      (if (res_null vr)
       then (ThrowException NullPointer state)
       else (getfield_obj t idx (tail (opstack h)) vx state)
      ) |
    _ ⇒ (AbortCode type_error state)
end end end.
```

La fonction Coq `GETFIELD` qui traduit cette sémantique fait appel à plusieurs fonctions auxiliaires.

La fonction `getfield_obj` effectue la recherche de l'instance et du champ demandé, puis confronte le type de ce champ au type avec lequel l'instruction a été appelée.

```
Definition getfield_obj := [t:type][idx:instance_field_idx]
[ops:(list valu)][vx:heap_idx][state:jcvm_state]
Cases (get_inst_from_heap (heap_f state) vx) of
None ⇒ (AbortMemory heap_error state) |
(Some u) ⇒
  Cases (Nth_elt (contents_i u) idx) of
  None ⇒ (AbortCap field_error state) |
  (Some nod) ⇒
    Cases nod t of
    (vRef _)               (Ref _)       ⇒
      (res_getfield ops nod state u) |
    (vPrim (vByte vnod))    (Prim Byte)   ⇒
      (res_getfield ops (vPrim (vShort vnod)) state u) |
    (vPrim (vBoolean vnod)) (Prim Byte)   ⇒
```

```
    (res_getfield ops (vPrim (vShort vnod)) state u) |
  (vPrim (vShort _))      (Prim Short)  ⇒
    (res_getfield ops nod state u) |
  (vPrim (vInt _))        (Prim Int)    ⇒
    (res_getfield ops nod state u) |
  _                        _             ⇒
    (AbortCode type_error state)
end end end.
```

La fonction `get_inst_from_heap` récupère dans le tas l'objet à la position donnée en argument et vérifie qu'il s'agit bien d'une instance. Comme beaucoup de fonctions dans la formalisation, il s'agit d'une fonction partielle : son type de retour est `(option type_instance)`.

```
Definition get_inst_from_heap := [hp:heap][vx:heap_idx]
Cases (Nth_func hp vx) of
None ⇒ (None ?) |
(Some nhp) ⇒
  Cases nhp of
  (Instance u) ⇒ (Some ? u) |
  _ ⇒ (None ?)
  end
end.
```

Enfin, la fonction `res_getfield` construit le résultat à partir de la pile d'opérande finale, de la valeur du champ, et évidemment, de l'état courant. Dans cette fonction, un dernier test est effectué par `test_exception_mo` pour assurer le mécanisme de sécurité de *Java Card* (cf 2.2.3) et une exception de sécurité peut alors être lancée.

```
Definition res_getfield :=
[ops:(list valu)][nod:valu][state:jcvm_state][inst:type_instance]
Cases (stack_f state) of
nil ⇒ (AbortCode state_error state) |
(cons h lf) ⇒
  Cases (test_exception_mo h inst) of
  (Some x) ⇒ (ThrowException x state) |
  None ⇒ (update_frame (push_opstack nod ops h) state)
  end
end.
```

3.4.4 Sémantique de invokevirtual

La sémantique de `invokevirtual` est légèrement plus complexe. Cette instruction appelle une méthode d'instance. Les $\texttt{nargs} - 1$ arguments de la méthode appelée et la référence à l'instance sont dépilés de la pile d'opérande. Un contexte d'exécution de méthode est créé contenant comme variables locales ces éléments dépilés. Son pointeur de programme est placé à 1. Placé

en haut de la pile de contextes, le nouveau contexte devient le contexte courant.

La fonction Coq INVOKEVIRTUAL effectue de nombreuses vérifications (détaillées en commentaire). Comme pour `getfield`, cette instruction peut lancer des exceptions, si la référence à l'instance est nulle, ou si les tests de sécurité effectués par `test_security_invokevirtual` ont échoué.

```
Definition INVOKEVIRTUAL :=
[nargs:nat][nm:class_method_idx][state:jcvm_state][cap:jcprogram]
Cases (stack_f state) of
nil ⇒ (AbortCode state_error state) |
(cons h lf) ⇒

  (* nargs must be greater than zero *)
  Cases nargs of
  O     ⇒ (AbortCode args_error state) |
  (S _) ⇒

    (* Extraction of the object reference (the nargsth element) *)
    Cases (Nth_func (opstack h) nargs) of
    None ⇒ (AbortCode opstack_error state) |
    (Some x) ⇒

      (* Tests if this element is a reference *)
      Cases x of
      (vNonInit _ _ _) ⇒ (AbortCode type_error state) |
      (vPrim _) ⇒ (AbortCode type_error state) |
      (vRef vr) ⇒

        (* Tests if the reference is null *)
        (if (res_null vr)
         then (ThrowException NullPointer state)
         else

          (* Extraction of the referenced object *)
          Cases (Nth_func (heap_f state) (vr2hi vr)) of
          None ⇒ (AbortMemory heap_error state) |
          (Some nhp) ⇒

            (* Get the corresponding class *)
            Cases (Nth_elt (classes cap) (get_obj_class_idx nhp)) of
            None ⇒ (AbortCap class_membership_error state) |
            (Some c) ⇒
              Cases (get_method c nm cap) of
              None ⇒ (AbortCap methods_membership_error state) |
              (Some m) ⇒

                (* Extraction of the list of arguments *)
                Cases (l_take nargs (opstack h)) (l_drop nargs (opstack h)) of
                (Some l) (Some l') ⇒

                  (* Security checking *)
```

```
                   (if (test_security_invokevirtual h nhp)
                    then
                    (Normal (Build_jcvm_state
                              (sheap_f state)
                              (heap_f state)
                              (cons (Build_frame (nil valu)
                                                 (make_locvars l (local m))
                                                 (method_id m)
                                                 (get_owner_context nhp)
                                                 (0)
                                                 (m_max_opstack_size m))
                                 (cons (Build_frame l'
                                                    (locvars h)
                                                    (method_loc h)
                                                    (context_ref h)
                                                    (p_count h)
                                                    (max_opstack_size h))
                              (tail (stack_f state))))))
                    else (ThrowException Security state)
                   ) |
                 _ _ ⇒ (AbortCode opstack_error state)
                 end
               end
             end
           end
        )
      end
    end
  end
end.
```

3.5 Gestion des erreurs et exceptions

Lors de l'exécution d'un programme, le déroulement normal des calculs peut être interrompu par un programme à la forme incorrecte ou une condition non réalisée d'un calcul. Dans le premier cas, il s'agit d'une erreur du programme (comme l'appel à l'instruction `pop` avec une pile vide, un typage incorrect, ...) qui conduit à la terminaison abrupte de l'exécution. Dans le second cas, le mécanisme des exceptions, présent dans beaucoup de langages de programmation récents et qui se traduit par un changement dans le flot de contrôle normal du programme, permet un gestion efficace de ces erreurs dynamiques d'exécution.

3.5.1 Les erreurs

À nouveau, l'origine des erreurs dans la machine virtuelle peut provenir de différentes sources. Il peut s'agir :

- d'une erreur du code (branchement à une adresse incorrecte, pile vide, ...) ;
- d'une incohérence du programme (classe inexistante, champ manquant, ...) ;
- d'une incohérence de la mémoire (objet non trouvé ou pas du type attendu).

Le signalement de ces erreurs dans la machine virtuelle s'effectue respectivement avec l'utilisation des fonctions `AbortCode`, `AbortCap` et `AbortMemory`.

Ces fonctions construisent un état anormal (`Abnormal`) à partir de la raison et de l'état donnés en argument. La raison, du type inductif `eReason` suivant, n'est présente que pour aider à rendre plus claire la cause de l'erreur :

```
Inductive eReason : Set :=
heap_error         : eReason |
sheap_error        : eReason |
opstack_error      : eReason |
checkcast_error    : eReason |
store_error        : eReason |
overflow_error     : eReason |
init_error         : eReason |
[...]
```

L'exécution ne peut continuer à partir d'un état anormal et la machine virtuelle termine.

3.5.2 Les exceptions

En *Java*, lors du lancement d'une exception, un objet *Java* est créé dont le type donne la raison de l'erreur. Un mécanisme de *rattrapage* de l'exception intervient alors et recherche parmi la pile des contextes d'exécution de méthode la méthode contenant le code à exécuter lorsqu'une telle exception est lancée. Le flot de contrôle est alors transmis au début de ce code et l'exécution continue.

Le lancement d'exceptions peut être réalisé ou bien directement par la machine virtuelle (dans le cas d'une division par zéro par exemple) ou par le programmeur (utilisation du mot-clé *Java* `throw` dans le code source, traduit vers le code octet `athrow`).

Lorsque la machine virtuelle lance d'elle-même une exception, l'état de retour de l'instruction exécutée a pour constructeur `ThrowException` et

comme arguments l'état courant et un terme du type `xLabel` indiquant l'exception à lancer :

```
Inductive xLabel : Set :=
Arithmetic             : xLabel |
ArrayStore             : xLabel |
ClassCast              : xLabel |
IndexOutOfBounds       : xLabel |
ArrayIndexOutOfBounds  : xLabel |
NegativeArraySize      : xLabel |
NullPointer            : xLabel |
Security               : xLabel .
```

Un objet java du type correspondant à l'exception voulue est alors créé dans le tas et la fonction `CatchException` (dont le comportement est décrit à la section 3.5.3, qui suit) est appelée.

Dans le cas du code octet athrow, l'objet représentant l'exception à lancer a déjà été créé par des codes octet précédant l'instruction et la fonction `CatchException` est, après quelques vérifications sur cet objet, appelée également.

3.5.3 Récupération d'exceptions

La récupération des exceptions est principalement un problème de recherche dans la pile des contextes d'exécution.

Chaque méthode possède sa liste de gestionnaires d'exceptions, il s'agit du champ `handler_list` de l'enregistrement `Method` (cf section 3.2.2). Un gestionnaire d'exception est décrit par le type `handler_type` :

```
Definition handler_type :=
  (bytecode_idx*bytecode_idx*cap_class_idx*bytecode_idx).
```

où les deux premiers index de code octet indiquent la portée du code dans la méthode (un bloc) où doit se trouver le pointeur d'exécution pour qu'une exception de type compatible avec la classe donnée en troisième composante puisse être gérée. La dernière composante indique l'emplacement du code (à l'intérieur du code de la méthode) où brancher dans le cas où l'exception est récupérée par ce gestionnaire.

Une méthode est alors en mesure de rattraper une exception si elle possède un gestionnaire d'exception tel que :

- sa portée inclut le pointeur d'exécution ;
- l'exception levée est une instance ou une instance d'une sous-classe de la classe gérée.

Lors du lancement d'une exception, le mécanisme de récupération d'exception commence par observer si la méthode courante (celle du haut de la pile de contextes d'exécution) peut gérer l'exception. Le cas échéant, l'exécution reprend à l'emplacement indiqué par le gestionnaire d'exception. Sinon, le contexte d'exécution courant est enlevé de la pile des contextes et l'algorithme recommence. Cela se traduit simplement par la fonction Coq suivante (une des seules de la formalisation à être exprimée par un point fixe) :

```
Fixpoint lookup_stack [s:stack] :
  Class→jcprogram→(option (bytecode_idx*stack)) :=
  [cl:Class][cap:jcprogram]
Cases s of
(* stack empty: exception uncaught *)
nil  ⇒ (None ?) |
(cons h lf) ⇒
  (* lookup in current frame *)
  Cases (lookup_frame cl h cap) of
  (* handler found in h: return branching point and current stack *)
  (Some u) ⇒  (Some ? (u,s)) |
  (* no handler found in frame h: continue lookup in lf *)
  None     ⇒ (lookup_stack lf cl cap)
  end
end.
```

On remarque qu'il s'agit d'une fonction partielle et que l'on peut se trouver dans une situation où aucun contexte ne gère l'exception. L'exécution du programme s'arrête alors.

On notera également qu'à l'intérieur d'un même contexte d'exécution, plusieurs gestionnaires peuvent gérer une exception. Le gestionnaire choisi est alors celui dont la portée du bloc gestionnaire est la moins large.

3.6 Exécution des programmes

Dès la phase de conception de la formalisation de la machine virtuelle, la possibilité d'exécuter un programme *Java Card* par l'intermédiaire de cette formalisation a été jugée comme un point crucial. En effet, cela permet de comparer le résultat de l'exécution de notre machine virtuelle à d'autres machines virtuelles, comme celle de Sun, et donc d'obtenir une formalisation robuste, réaliste et sûre.

Dans la chaîne des outils disponibles pour pouvoir exécuter avec notre formalisation un programme *Java Card* manquait un convertisseur pour traduire les fichiers compilés *Java Card* représentant le programme vers une expression Coq de type `jcprogram`. Pour atteindre ce but, nous avons utilisé les capacités du programme *JCVM Tools*.

3.6.1 Conversion des programmes

Avant de pouvoir être chargé, avec la formalisation, le code source d'un programme *Java Card* doit suivre les étapes décrites dans la figure 3.3.

En premier, le code source du programme, écrit en langage *Java* mais n'utilisant que des fonctionnalités de *Java Card*, est compilé en un fichier *class* par un compilateur standard (celui de Sun par exemple).

Ensuite, les fichiers *class* composant le programme sont donnés aux programmes *Java Card class file converter* et *Java Card Cap file builder* de Sun, pour produire un fichier *capfile* , représentation finale d'un programme *Java Card* prêt à être chargé sur une carte à puce.

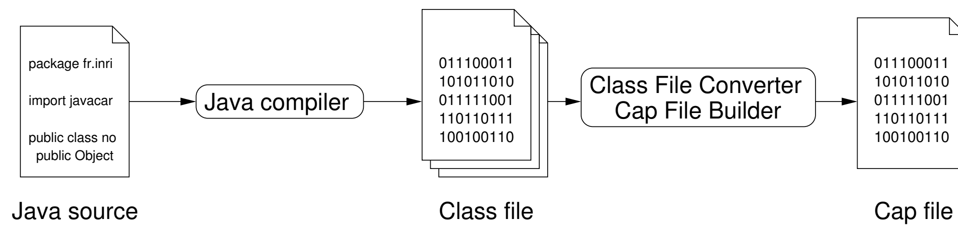

FIG. 3.3 – *Compilation d'un programme* Java Card.

Pour être utilisé avec notre formalisation, le fichier *capfile* doit être converti en un fichier COQ. En effet, COQ est incapable de pouvoir lire directement un fichier binaire. Cette étape est réalisée par l'outil *JCVM Tools* décrit dans la section 3.6.2 et dans la figure 3.4. Le fichier COQ produit est compilé avec le compilateur COQ, puis chargé, avec la formalisation, dans l'interpréteur de commande.

Il est également possible d'utiliser COQ pour extraire la formalisation et le programme vers OCAML. L'exécution du programme est alors plus rapide et n'est plus bornée par un nombre fixé de pas d'exécution.

3.6.2 Le JCVM Tools

Le *JCVM Tools* est un logiciel écrit en *Java* et développé par Bernard Paul SERPETTE. Bien qu'il dispose de nombreuses fonctionnalités, comme la navigation à travers les données du fichier *capfile* ou la possibilité à la manière d'un déboggueur d'exécuter un programme et d'observer le contenu de la mémoire, nous nous intéresserons ici qu'à la partie de ce logiciel permettant de traduire des fichiers *capfile* vers une expression (et un fichier) COQ.

Notre machine virtuelle fonctionne avec une représentation structurée

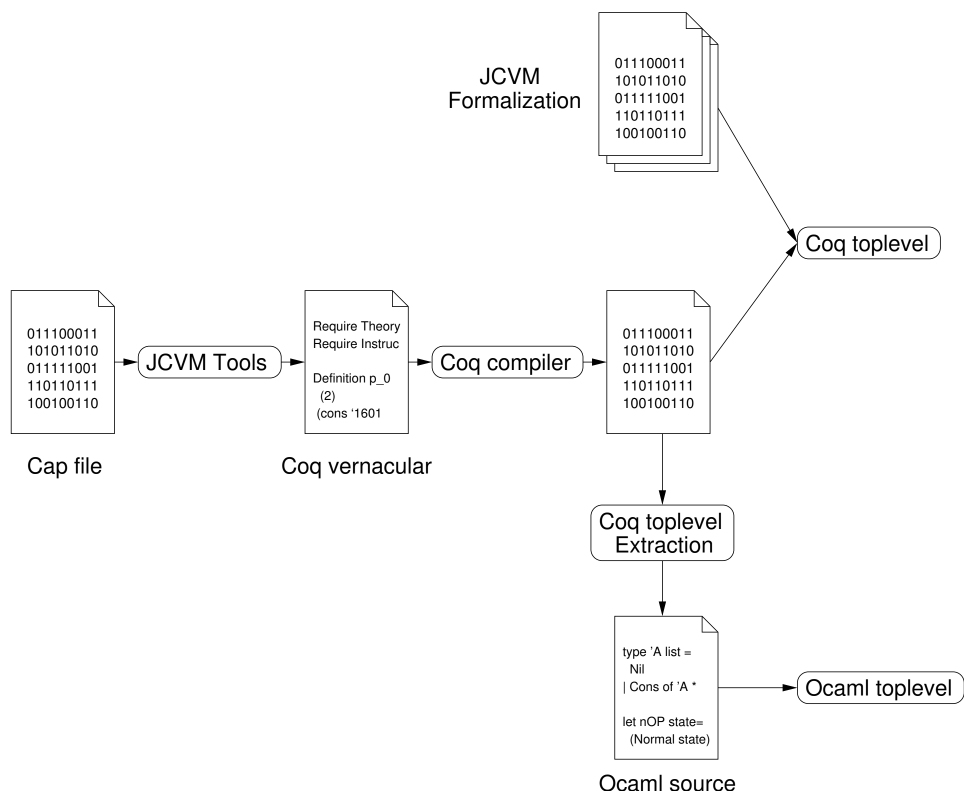

FIG. 3.4 – *Transformation d'un programme* Java Card.

du programme (décrite à la section 3.2). Il serait extrêmement fastidieux de travailler directement dans Coq avec le programme sous la forme binaire des fichiers *capfile*. La représentation structurée est alors réalisée par le *JCVM Tools* qui joue alors un rôle à peu près similaire à celui d'un compilateur en effectuant des phases de liaison et de résolution de la zone des constantes. Cette zone des constantes est un tableau de données variées. Ces données peuvent être une chaîne de caractères, un nom de classe ou d'interface, une référence vers un champ ou une méthode, une valeur numérique. À l'intérieur du fichier *classfile*, les références à des chaînes de caractères, à des classes, champs ou méthodes ne peuvent se faire que par l'intermédiaire d'un index dans la zone des constantes.

Les principales transformations effectuées par le *JCVM Tools* sont ainsi :

- La résolution complète et l'élimination de la zone des constantes (constant pool). Pour les codes octet par exemple, et avec la factorisation dans notre formalisation de plusieurs codes octet en un seul, cela peut conduire à des instructions avec des opérandes différentes de la spécification de Sun.
- La traduction des classes, méthodes et interfaces vers les structures correspondantes de notre formalisation.
- D'autres transformations mineures comme la traduction des adresses de branchement relatives en adresses absolues (des entiers naturels).

Par ces transformations, le *JCVM Tools* assure un certain nombre d'invariants, principalement structurels. Ces invariants sont requis soit par le vérificateur de code octet (voir description section 5.1) sur le fichier *capfile*, soit par la spécification des instructions [121, Chapitre 7] (emploi du mot must dans les spécifications).

Concernant les invariants sur les instructions, le *JCVM Tools* assure, si la phase de résolution s'est bien déroulée, que les vérifications structurelles dans les spécifications des instructions sont réalisées. Intéressons-nous par exemple au code octet `invokevirtual`. Les spécifications de Sun requièrent que l'index fourni en opérande soit une référence valide vers une classe et une méthode virtuelle de la zone des constantes. Le *JCVM Tools* traduit ces références vers les structures de programme utilisées par la formalisation en Coq et assure ainsi que les références sont valides. La méthode doit être un constructeur de classe ou d'interface, ce que vérifie également le *JCVM Tools* et qui n'a plus à être fait par la formalisation. Enfin, le *JCVM Tools* vérifie les contraintes sur les attributs de méthodes (publiques, protégées ou privées) qui n'ont plus alors à figurer dans l'enregistrement Coq `Method` ni à être testée par la formalisation. Pour les codes octet de branchement, il assure que l'adresse cible est bien parmi les instructions de la méthode courante. Nous nous éloignons ainsi, comme le souligne [80], légèrement du modèle

d'exécution présent dans les cartes à puce. Pourtant, ces opérations ne sont que des préalables à l'exécution (elles sont vérifiées par le vérificateur de code octet) et ne changent en rien le comportement calculatoire des programmes.

Sur les programmes générés en sortie, le *JCVM Tools* assure également certaines contraintes sémantiques, comme le fait qu'à l'intérieur d'une méthode les gestionnaires d'exception apparaissent bien dans l'ordre où ils ont été déclarés dans le programme. Dans le cas contraire, le mécanisme de gestion d'exception pourrait sélectionner un gestionnaire différent.

3.6.3 Exemple de conversion

Pour illustrer la phase de conversion, nous partons d'un programme *Java Card* simple et plus particulièrement d'une classe *Java* implémentant un tri par sélection.

```java
package demo;

public class sort {
    public comparable[] values;

    public sort(comparable[] vs) {
        values = vs;
    }

    public short min(short i) {
        short r = i;
        comparable min = values[i];
        for(short j=(short)(i+1) ; j<values.length; j++) {
            comparable o = values[(short) j];
            if(min.greater(o)) {
                min = o;
                r = (short) j;
            }
        }
        return(r);
    }

    public void dosort() {
        for(short i=0; i<values.length; i++) {
            short mini = min(i);
            comparable save = values[mini];
            values[mini] = values[i];
            values[i] = save;
        }
    }
}
```

Une fois les étapes de conversion passées, on obtient un fichier Coq contenant, entre autres, la définition suivante de la méthode `min`. On ob-

servera la suite du code octet formant le corps de la méthode et d'autres informations comme, sur la quatrième ligne, sa signature.

```
Definition m_18 := (Build_Method    (* min *)
  ( false
  false
  (cons (Prim (Short)) (nil type)), (Prim (Short)))
  (4)
  (* 0 *)  (cons (load (Prim (Short)) (1))
  (* 1 *)  (cons (store (Prim (Short)) (2))
  (* 2 *)  (cons (getthisfield (Ref (Ref_instance (0))) (0))
  (* 3 *)  (cons (load (Prim (Short)) (1))
  (* 4 *)  (cons (aload (Ref (Ref_instance (0))))
  (* 5 *)  (cons (store (Ref (Ref_instance (0))) (3))
  (* 6 *)  (cons (load (Prim (Short)) (1))
  (* 7 *)  (cons (push (Short) '1')
  (* 8 *)  (cons (arith opAdd (Short))
  (* 9 *)  (cons (store (Prim (Short)) (4))
  (* 10 *)  (cons (goto (24))
  (* 11 *)  (cons (getthisfield (Ref (Ref_instance (0))) (0))
  (* 12 *)  (cons (load (Prim (Short)) (4))
  (* 13 *)  (cons (aload (Ref (Ref_instance (0))))
  (* 14 *)  (cons (store (Ref (Ref_instance (0))) (5))
  (* 15 *)  (cons (load (Ref (Ref_instance (0))) (3))
  (* 16 *)  (cons (load (Ref (Ref_instance (0))) (5))
  (* 17 *)  (cons (invokeinterface (2) (0) (0))
  (* 18 *)  (cons (if_cond (Zeqb) (23))
  (* 19 *)  (cons (load (Ref (Ref_instance (0))) (5))
  (* 20 *)  (cons (store (Ref (Ref_instance (0))) (3))
  (* 21 *)  (cons (load (Prim (Short)) (4))
  (* 22 *)  (cons (store (Prim (Short)) (2))
  (* 23 *)  (cons (inc (Short) '1' (4))
  (* 24 *)  (cons (load (Prim (Short)) (4))
  (* 25 *)  (cons (getthisfield (Ref (Ref_instance (0))) (0))
  (* 26 *)  (cons (arraylength)
  (* 27 *)  (cons (if_scmp_cond (Zltb) (11))
  (* 28 *)  (cons (load (Prim (Short)) (2))
  (* 29 *)  (cons (return (Prim (Short)))
  (nil Instruction))))))))))))))))))))))))))))))
  false
  (nil handler_type)
  (6)
  (18)
  (13) ).
```

Cette méthode `m_18` est ensuite utilisée dans la définition de la classe `sort` (parmi les méthodes publiques) :

```
Definition c_13 := (Build_Class    (* sort *)
  (cons (0) (nil nat))
  (cons m_2 (cons m_17 (cons m_18 (nil Method))))
  (nil Method)
  (nil ?)
  (cons (Ref (Ref_array ((Ref (Ref_interface (0))))))
```

```
  (nil type))
  p_1
  (13) ).
```

Enfin, la classe est reprise dans la définition du programme. Les nombreuses autres classes proviennent des classes *Java Card* standard comme `Object`, `Shareable`.

```
Definition main := (Build_jcprogram
   (cons c_0 (cons c_1 (cons c_2 (cons c_3
   (cons c_4 (cons c_5 (cons c_6 (cons c_7
   (cons c_8 (cons c_9 (cons c_10 (cons c_11
   (cons c_12 (cons c_13 (cons c_14
     (nil Class)))))))))))))))
   (cons m_boot (cons m_1
   (cons m_2 (cons m_3 (cons m_4 (cons m_5
   (cons m_6 (cons m_7 (cons m_8 (cons m_9
   (cons m_10 (cons m_11 (cons m_12 (cons m_13
   (cons m_14 (cons m_15 (cons m_16 (cons m_17
   (cons m_18 (cons m_19 (cons m_20
     (nil Method)))))))))))))))))))))
   (cons i_0 (nil Interface))
   (nil type).
  ).
```

3.6.4 Utilisation des APIs

Les Application Programmer Interface (API)s sont des collections de librairies offertes au programmeur pour faire des requêtes, par l'intermédiaire de l'appel de méthodes, au système d'exploitation ou à une autre application. Les APIs *Java* sont organisées en paquettages de code déjà écrit qui traitent de fonctionnalités spécifiques. Par exemple, les APIs *Java Card* [119] du paquettage `javacard.crypto` donnent l'accès à des fonctions de cryptographie pour les cartes à puce.

En plus de la machine virtuelle *Java Card*, l'environnement d'exécution *Java Card* inclut normalement une implémentation des APIs *Java Card*. Cette implémentation est requise pour exécuter les programmes *Java Card* qui reposent sur ces APIs. Dans cette section, nous allons décrire une méthodologie simple pour étendre notre modèle avec les APIs et illustrer son fonctionnement sur les APIs impliquées dans le partage sécurisé d'objets.

La plupart des APIs peuvent être converties sans problème avec le Cap file converter puisqu'elles reposent sur du code *Java Card* standard. On obtient alors un fichier *capfile* qui est chargé par le *JCVM Tools* avec le fichier *capfile* représentant le programme utilisant ces APIs. Enfin, la zone des constantes est résolue et le *JCVM Tools* produit un fichier Coq contenant à la fois les APIs et le programme.

Cependant, cette conversion échoue avec les APIs utilisant les méthodes natives.

3.6.5 Les méthodes natives

Certaines APIs, principalement les APIs de `javacard.framework` et `com.sun.javacard.impl`, utilisent des méthodes natives. Les méthodes natives sont des méthodes dont le code ne peut pas être écrit en *Java* mais uniquement dans le langage avec lequel la machine virtuelle a été conçue (Coq dans notre cas). Elles ont accès alors à l'environnement-même dans lequel la machine virtuelle s'exécute.

De telles méthodes sont déclarées en *Java* avec le mot-clé `native`. Cette propriété est conservée dans le fichier *classfile* mais le convertisseur *Java Card* ne peut convertir des fichiers *classfile* avec des méthodes natives (le format de fichier *capfile* ne les prévoit pas). On ne peut donc obtenir de fichiers *capfile* pour ces APIs.

Afin de bénéficier facilement des APIs utilisant des méthodes natives `com.sun.javacard.impl`, c'est-à-dire disposer directement du code non-natif écrit en *Java* et n'avoir qu'à modifier les méthodes natives, il faut modifier le code source des APIs.

Partout où une méthode est déclarée native, on retire le mot-clé et on donne comme corps l'appel à une méthode avec la même signature appartenant à une classe spécifique que l'on aura créée,

Par exemple, à l'intérieur du fichier `JCSystem.java` du paquettage `com.sun.javacard.framework`, on a la déclaration :

```
public static native byte getCurrentContext();
```

que l'on remplace par :

```
public static byte getCurrentContext() {
  return (DummyNatives.getCurrentContext()); }
```

et à l'intérieur de la classe `DummyNatives`, on déclare

```
public static byte getCurrentContext() {
  return ((byte) 0); }
```

pour que le fichier compile, mais ce corps ne sera pas utilisé. Au moment de l'exécution par notre machine virtuelle, la méthode correspondante, écrite en Coq, sera appelée à la place.

On peut noter que cette méthodologie est aussi employée, mais non documentée, dans le Java Card Workstation Development Environment

(JCWDE), l'environnement de test de programmes *Java Card* fourni par Sun.

Les APIs modifiées de cette manière peuvent maintenant être converties en un fichier *capfile* standard. Le *JCVM Tools* est ensuite capable de reconnaître les méthodes natives modifiées (par l'appel à une méthode appartenant à la classe `DummyNatives`), et de convertir cet appel vers un code octet spécial, `invokenative`, de notre formalisation Coq. Une opérande du code octet, un index, identifie la méthode appelée.

Il reste finalement à fournir pour chaque méthode native leur implémentation en Coq. Bien que cette méthodologie ait été testée, l'implémentation des APIs, qui représente un important travail de développement, n'a pas actuellement été réalisée.

3.6.6 Le paquettage `javacard.framework`

Le bénéfice d'implémenter les méthodes natives du paquettage `javacard.framework` (au nombre de 47) serait d'obtenir un environnement d'exécution *Java Card* complet et d'être capable d'exécuter n'importe quel programme *Java Card*.

En particulier, le paquettage `javacard.framework` définit les méthodes relatives aux APDUs, seul lien de communication entre la carte à puce et l'extérieur. Dans le cadre de preuves de programmes *Java Card*, la formalisation des APDUs constituerait un ajout essentiel et relativement simple (la notion d'état contiendrait alors deux champs supplémentaires pour les tampons d'entrée/sortie d'APDUs).

D'autres notions en revanche, comme la formalisation du mécanisme de transactions des APIs de `javacard.framework`, semblent présenter moins d'intérêt, d'une part en raison de la complexité de leur mise en œuvre ; d'autre part en raison de la portée des preuves reposant alors sur une implémentation particulière du mécanisme de transaction qui ne serait pas nécessairement celle en place sur la carte à puce.

Finalement, et avec la formalisation du paquettage `javacard.framework`, le point de départ de notre formalisation pour exécuter un programme serait alors la méthode `main` de la classe `Dispatcher`, *i.e.* le point de départ standard d'une carte à puce *Java Card* après une réinitialisation.

3.7 Conclusion

Nous avons abordé dans ce chapitre la formalisation Coq d'une machine virtuelle *Java Card* exécutable réalisant dynamiquement les vérifications de type proposées par les spécifications de Sun. Pour faire écho à l'introduction de ce chapitre, les critères suivants sur la réalisation de la plate-forme *Java Card* ont été satisfaits :

Complétude. Notre machine virtuelle peut-être considérée comme complète dans la mesure où elle donne une sémantique à l'ensemble des instructions *Java Card*. De plus, elle prend en compte les aspects les plus importants (ceux dont la sécurité dépend) de la plate-forme comme le pare-feu ou les mécanismes de partage d'objets. Nous proposons d'autre part une méthodologie pour intégrer les aspects manquant comme les méthodes natives, sans pour autant proposer une implémentation de celles-ci (il s'agit d'un effort important, mais sans difficultés d'ordre conceptuel) puisque, dépendantes de la plate-forme matériel finale de la machine virtuelle, elles apporteraient en Coq peu voire rien à la vérification globale de la plate-forme.

Réalisme et précision. En fournissant grâce au *JCVM Tools* un outil pour convertir un programme *Java Card* sous forme de fichier *capfile* et en disposant d'une formalisation exécutable (sous Coq comme sous Ocaml), nous sommes en mesure d'exécuter tout programme *Java Card* (mais sans appel à des méthodes natives). Les traces d'exécution fournies permettent de comparer les résultats obtenus à une machine virtuelle de référence, comme celle de Sun.

Adéquation à la preuve. En n'utilisant qu'un sous-ensemble limité des capacités de spécifications de Coq (types inductifs, enregistrements, définition fonctionnelle et filtrage), nous limitons les schémas possibles de preuves. Nous étudierons dans les chapitres suivants les avantages de ce type de spécification pour leur manipulation et la construction de preuve. Enfin, un point non abordé ici concernerait la possibilité de raisonner sur les traces d'exécution fournies par la formalisation. Des procédés tels que la logique temporelle offrent des outils efficaces pour le raisonnement sur ces traces d'exécution.

Dans un autre registre, Gemplus a pu étudier la formalisation dans le cadre d'une évaluation EAL 7 [80]. Leurs conclusions valident la méthodologie employée, regrettent quelques écarts avec les spécifications Sun (qui ont pu être corrigés depuis 2001) et vont dans le sens d'une intégration plus poussée dans Coq des fonctions *Java Card* (comme celle réalisées par le *JCVM Tools*).

Travaux futurs La méthodologie présentée dans ce chapitre semble aboutie. Un important travail d'implémentation resterait encore à réaliser pour les méthodes natives, sans bénéfice évident du point de vue des preuves. L'extension des travaux ci-dessus irait plutôt dans le sens de l'intégration d'autres composantes de l'architecture *Java Card*, comme le transformateur de fichiers *capfile* ou de l'éditeur de lien tel que le *JCVM Tools*. Les travaux de Denney et Jensen [51] sur les optimisations, et leur correction, du code octet dans la transformation d'un fichier *classfile* à un fichier *capfile* répondent à cette attente. Il serait intéressant de considérer leur intégration dans notre formalisation, d'autant plus que ces travaux ont également été réalisés dans Coq.

Une autre extension (orthogonale cette fois) pouvant être considérée serait d'étendre la machine virtuelle au langage *Java* (avec lequel *Java Card* partage un grand nombre d'instructions). La gestion des nombres flottants serait nécessaire mais des formalisations de ceux-ci existent déjà en Coq [50]. Les problèmes principaux serait générés par les fonctionnalités de l'environnement d'exécution comme le chargement dynamique de classe [44, 70], le ramasse-miettes [30], les fils d'exécutions [118].

CHAPITRE 4

Abstractions de la machine virtuelle

La machine virtuelle définie dans la partie précédente nous offre un modèle réaliste pour l'exécution des programmes et se présente, par son intégration à l'assistant de preuve Coq, facilement utilisable pour le raisonnement.

Cependant, avant de pouvoir raisonner sur la machine virtuelle et les programmes, il est nécessaire de réduire l'espace de valeurs possibles au moyen d'abstractions [49]. En effet, les espaces de valeurs considérés, souvent infinis (comme l'espace des valeurs de la machine virtuelle), ne permettent pas de procéder à des analyses sur ces dernières en un temps fini.

Nous commencerons donc ce chapitre par présenter la méthodologie employée pour vérifier des propriétés de programme à partir la réalisation d'une machine virtuelle abstraite selon les propriétés à assurer. Cette machine virtuelle constituera alors la base principale d'un vérificateur de code octet (cf chapitre 5) dont le rôle sera d'assurer qu'un programme vérifie la propriété donnée.

Nous illustrerons la construction de machines virtuelles abstraites par les abstractions de type et de valeur [11]. Rappelons que dans la machine virtuelle de la partie précédente les valeurs transportaient dans un type inductif une information numérique et un type. Suivant les travaux de Cohen [45], nous qualifirons cette machine de *défensive* [1] car elle vérifie les types des valeurs lors des calculs. Nous construirons alors une machine machine virtuelle *offensive* en ne conservant que l'information numérique des valeurs

1. Gerwin Klein préfère dans sa thèse [76] le qualificatif *agressive*.

de la machine virtuelle et une machine virtuelle *abstraite* en ne conservant que la partie type.

Machine virtuelle	Défensive	Offensive	Abstraite
Valeur	Type $*$ Num	Num	Type

Enfin, nous présenterons l'outil *Jakarta* dont le rôle est d'automatiser les tâches de construction des machines virtuelles abstraites.

4.1 Méthodologie pour la vérification de propriétés

Nous nous plaçons dans le cas général d'une propriété P à vérifier sur une machine virtuelle. P peut-être instanciée à la sûreté du typage comme dans notre cas, ou la vérification du contrôle de ressources (disponibilité de la mémoire et de l'unité de calcul), l'initialisation des objets (chaque objet doit être initialisé par l'appel de son constructeur avant son utilisation), ou toute autre propriété.

Le moyen le plus simple de garantir la propriété P reste de la vérifier dynamiquement au cours de l'exécution et de lever, en cas de violation de P, une erreur. Ce type de vérification est réalisé par une machine dite défensive pour P (ou P-défensive). Cependant, cette approche est peu efficace en terme de rapidité d'exécution.

Dès lors que la propriété ne dépend pas trop fortement des valeurs de l'exécution et qu'il est possible d'obtenir une abstraction raisonnable (c'est-à-dire avec un espace de valeur fini) de cette propriété, il devient concevable de réaliser la vérification de manière statique. Cette opération est effectuée par un vérificateur de code octet pour P (ou P-vérificateur de code octet). Ce dernier est construit à partir d'une machine, dite P-abstraite, ne réalisant au cours de l'exécution que les opérations nécessaires à la vérification de P, à l'exclusion des autres calculs.

Le vérificateur de code octet a alors la charge d'assurer que tout programme analysé s'exécutera sans erreurs liées à P sur la machine virtuelle défensive. En conséquence, les tests réalisés par la machine virtuelle défensive deviennent inutiles et peuvent être retirés. On obtient alors une machine virtuelle P-offensive, plus rapide à l'exécution, plus légère en occupation mémoire, et aussi sûre que la machine virtuelle P-défensive si le programme a passé la vérification de code octet.

On peut ainsi résumer cette distinction sur les machines virtuelles entre calculs et vérification de propriété par l'équation :

$$P\text{-défensive} = P\text{-offensive} + P\text{-abstraite}$$

La relation forte entre ses trois machines virtuelles conduit au fait que les machines virtuelles offensive et abstraite peuvent être obtenues de manière presque systématique à partir de la machine virtuelle défensive. Si la machine virtuelle offensive semble la plus facile à obtenir, comme cela est confirmé pour la vérification de type, la machine virtuelle abstraite peut également être dérivée par des procédés d'abstraction au prix d'une intervention un peu plus importante pour guider l'abstraction. Ces techniques de dérivation, par l'intermédiaire de l'outil *Jakarta*, sont décrites dans la section suivante 4.4.

Pour le vérificateur de code octet maintenant, nous fournissons dans le chapitre 5 un cadre générique pour aider à sa réalisation. Il repose principalement sur la donnée d'une machine virtuelle abstraite, telle que celle dérivée par *Jakarta*. Interviennent ensuite un certain nombre de preuves (appelées validations croisées) assurant une relation en termes d'exécution entre la machine virtuelle défensive et ses abstractions, traduisant ainsi la correction des abstractions réalisées. La figure suivante résume les opérations menées sur la machine virtuelle défensive.

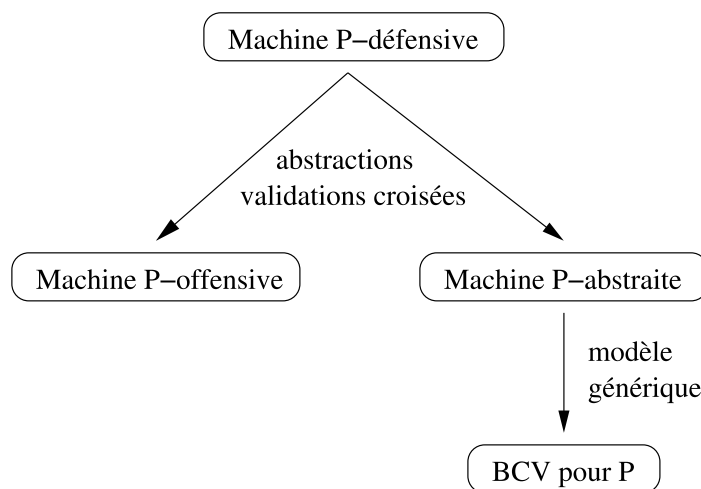

FIG. 4.1 – *Méthodologie pour la vérification de P.*

Notons finalement qu'il apparaît essentiel d'après cette figure que la formalisation de la machine virtuelle défensive corresponde aux spécifications de départ puisque toutes les autres opérations en dépendent. Notons également que grâce à une automatisation poussée des opérations réalisées, une modification sur la machine virtuelle défensive se reporte facilement sur les autres machines.

4.2 La machine virtuelle offensive

Appliqué à une abstraction de la machine virtuelle défensive selon les types des valeurs, la machine virtuelle offensive, telle que présentée à la section précédente, n'effectue plus de tests sur ces types.

Du point de vue de l'exécution, cette machine virtuelle est similaire à une machine que l'on pourrait trouver sur une carte à puce. Elle effectue les calculs plus efficacement et plus rapidement car elle a moins de tests à effectuer. Combinée avec un vérificateur de bytecode (voir chapitre 5) qui assure que le typage du programme est correct, elle apporte la même sécurité que la machine virtuelle défensive.

Nous allons décrire les changements que cela implique sur les représentations de l'environnement d'exécution et des instructions.

Notons enfin que nous allons être amenés à redéfinir pour les machines virtuelles offensive et abstraite certains types de données de la machine défensive. Pour des raisons de réutilisation de code, lorsque ces types sont redéfinis, leur nom est conservé. Pour résoudre le cas échéant toute ambiguïté sur la définition réelle utilisée, Coq fournit le mécanisme des noms qualifiés (voir section 5.2.1).

4.2.1 Représentation de l'environnement d'exécution

L'environnement d'exécution de la machine virtuelle offensive reste quasiment identique à celui de la machine virtuelle défensive, influencé simplement par la nouvelle définition des valeurs.

Les valeurs

Dès lors que la notion de valeur de la machine virtuelle ne transporte plus qu'une information numérique, il n'est plus possible de distinguer le type Coq (`nat` ou `Z`) le plus adapté pour représenter cette valeur en fonction de son type. On choisit le type le plus général, celui des entiers relatifs, et on définit donc :

```
Definition valu := Z.
```

On sera ainsi parfois amenés à utiliser des coercions entre `nat` et `Z` pour les fonctions manipulant ou rendant spécifiquement un des deux types.

Les états

Les notions d'état (tas, contexte d'exécution de méthode, état) restent inchangées par rapport à la machine défensive, mais ils utilisent maintenant la nouvelle définition de valeur.

Notons qu'une présentation modulaire, l'état utilisant une représentation abstraite sous forme de module des valeurs instanciée *a posteriori* aurait permis d'éviter de dupliquer les notions d'état. Cependant le système de module n'est arrivé dans Coq que très dernièrement. De plus, cela complique singulièrement la dépendance des fichiers et l'écriture de la formalisation elle-même.

4.2.2 Représentation des instructions

La représentation des instructions de la machine virtuelle offensive est là encore très proche de celle de la machine virtuelle défensive. Conceptuellement, il s'agit de supprimer les tests des instructions portant sur le type des valeurs. Ces tests-là sont facilement identifiables dans la machine virtuelle défensive car en cas d'échec ils aboutissent à l'appel `(AbortCode type_error state)`.

Notons que le vérificateur de code octet présenté au chapitre 5 garantit bien que si la vérification est passée alors il n'y a effectivement pas d'erreurs de typage et qu'il est donc valide de retirer ces tests. Cependant, le vérificateur de code octet garantit également d'autres propriétés comme l'accessibilité aux éléments voulus dans la pile de contextes d'exécution ou dans la pile d'opérandes. Toutefois, ces tests d'accessibilité demeurent toujours présents dans la machine offensive car ils ne sont pas concernés par l'abstraction réalisée sur le typage et car le traitement des fonctions partielles (comme les accesseurs) dans Coq oblige malgré tout à considérer tous les cas.

Nous allons maintenant illustrer à partir des instructions représentatives `if_acmp` et `getfield` décrites précedemment (section 3.4.3) les changements que nous sommes amenés à effectuer.

Sémantique de if_acmp

La version offensive de l'instruction `if_acmp` conduit à la fonction Coq suivante :

```
Definition IF_ACMP_COND :=
[oper:opcmp][branch:bytecode_idx][state:jcvm_state]
Cases (stack_f state) of
```

```
nil ⇒ (AbortCode state_error state) |
(cons h lf) ⇒
  Cases (head (opstack h)) of
  None ⇒ (AbortCode opstack_error state) |
  (Some v2) ⇒
    Cases (head (tail (opstack h))) of
    None ⇒ (AbortCode opstack_error state) |
    (Some v1) ⇒
       (update_frame (res_pc2
         (res_acompare2 oper v1 v2)
          branch h)
        state)
end end end.
```

où les fonctions `res_acompare2` et `res_pc2` sont similaires à leurs versions défensives.

Par rapport à la version de la page 58, on remarquera que les tests portant sur les types ont disparu et que par conséquence, l'instruction n'est plus susceptible de lancer une erreur de typage `type_error`.

Sémantique de getfield

La version offensive de l'instruction `getfield` conduit à la fonction Coq suivante :

```
Definition GETFIELD := [t:type][idx:instance_field_idx]
[state:jcvm_state][cap:jcprogram]
Cases (stack_f state) of
nil ⇒ (AbortCode state_error state) |
(cons h lf) ⇒
  Cases (head (opstack h)) of
  None     ⇒ (AbortCode opstack_error state) |
  (Some x) ⇒
    (if (res_null x)
     then (ThrowException NullPointer state)
     else (getfield_obj t idx (tail (opstack h)) x state)
    )
end end.
```

On remarque par rapport à la version défensive de la page 59 que le test de typage sur `x` a été supprimé. Le test `res_null` porte alors maintenant sur un terme de type `valu` et non plus `valu_ref` et consiste simplement à tester si `x` est égal à 0. De même `getfield_obj` reçoit maintenant un argument de type `valu` et non plus `heap_idx`. Le corps de cette dernière fonction est ainsi modifié en :

```
Definition getfield_obj := [t:type][idx:instance_field_idx]
[ops:(list valu)][x:valu][state:jcvm_state]
Cases (get_inst_from_heap (heap_f state) (absolu x)) of
```

```
None ⇒ (AbortMemory heap_error state) |
(Some u) ⇒
  Cases (Nth_elt (contents_i u) idx) of
  None ⇒ (AbortCap field_error state) |
  (Some nod) ⇒ (res_getfield ops nod state u)
end end.
```

où `absolu` renvoie, sous la forme d'un entier naturel, la valeur absolue d'un entier relatif.

4.3 La machine virtuelle abstraite

La machine virtuelle abstraite peut être considérée comme la contre-partie de la machine virtuelle offensive. Dans cette machine, les calculs ne sont donc pas effectués, seul les types statiques des valeurs manipulées sont vérifiés.

Du point de vue de l'exécution, cette machine virtuelle présente donc peu d'intérêt. En revanche, elle est une composante essentielle du vérificateur de bytecode, donc le but est justement de vérifier que les valeurs sont utilisées avec le bon type.

4.3.1 Représentation de l'environnement d'exécution

Par rapport à la machine virtuelle offensive présentée ci-dessus, l'environnement d'exécution subit beaucoup plus de changements.

Les valeurs

Les valeurs de la machine virtuelle abstraite sont les types des valeurs de la machine virtuelle défensive. Ils sont définis de manière similaire aux valeurs de cette dernière. On commence par les valeurs abstraites primitives :

```
Inductive valu_prim : Set :=
  vByte          : valu_prim |
  vShort         : valu_prim |
  vInt           : valu_prim |
  vBoolean       : valu_prim |
  vReturnAddress : bytecode_idx → valu_prim.
```

Par rapport à la version défensive, les valeurs numériques ont été enlevées, sauf pour le type `ReturnAddress`. En effet, la valeur associée à ce type est essentielle pour les sous-routines, elle conditionne le flot de contrôle

pour l'instruction `ret`. Cependant, cela ne pose pas de problème pour l'abstraction, cette valeur est statique et déterminée à partir du compteur de programme lors de l'appel à l'instruction `jsr`.

Les valeurs des références sont définies comme suit :

```
Inductive valu_ref : Set :=
  vRef_null     : valu_ref |
  vRef_array    : type → valu_ref |
  vRef_instance : class_idx → valu_ref.
```

où la valeur numérique de type `heap_idx` a disparu (nous verrons que le tas a alors aussi naturellement disparu puisqu'il n'y a plus de valeurs pour y accéder).

Enfin, on utilise les définitions précédentes pour construire le type Coq des valeurs abstraites :

```
Inductive valu : Set :=
  vPrim    : valu_prim → valu |
  vRef     : valu_ref → valu |
  vNonInit : class_idx → bytecode_idx → valu.
```

Là encore, la valeur des objets non initialisés `vNonInit` ne fait plus de référence au tas.

Les contextes d'exécution de méthode

Les contextes d'exécution de méthode utilisent la définition abstraite des valeurs :

```
Record frame : Set := {
  (* operand stack *)
  opstack       : (list valu);

  (* local variables *)
  locvars       : (list (option valu));

  (* location of the method *)
  method_loc    : cap_method_idx;

  (* program counter *)
  p_count       : bytecode_idx;

  (* max opstack size *)
  max_opstack_size : nat
}.
```

On notera que la référence au paquettage auquel appartient le contexte n'apparaît plus, cette information étant d'ordre dynamique.

Les états

La notion d'état pour la machine virtuelle abstraite est très simplifiée. Le tas n'existe plus, il est en effet constitué d'objets dynamiques. Les informations sur les valeurs que les objets transportent normalement seront extraites à partir du type des valeurs de références manipulées censées pointer vers un objet. Ces types contiennent effectivement pour les tableaux le type de ses éléments et pour les instances une indication de classe à partir de laquelle on peut déterminer le type statique des champs.

Les types du tas statique sont extraits du champ `sheap_type` des programmes.

Notons aussi que la machine virtuelle abstraite opére méthode par méthode : il n'y a plus d'appels de méthode et de création de nouveau contexte. En effet, la méthode appelée dépend pour les codes octet `invokevirtual` et `invokeinterface` du type dynamique de l'objet sur la pile, qu'il n'est plus possible d'obtenir.

L'état est donc limité à un seul contexte d'exécution de méthode :

```
Definition jcvm_state := frame.
```

Enfin, les états de retour des instructions ont seulement deux constructeurs, celui correspondant aux exceptions n'apparaît plus car ces dernières ne peuvent être lancées que de manière dynamique :

```
Inductive returned_state: Set :=
Normal   :          jcvm_state→returned_state |
Abnormal : eLabel→jcvm_state→returned_state.
```

Notons que si les exceptions ne peuvent plus être lancées par la machine virtuelle abstraite, le code des gestionnaires d'exception reste évidemment présent dans la représentation des programmes qui ne change pas. Ce code sera analysé séparément par le vérificateur de code octet.

4.3.2 Représentation des instructions

L'abstraction des types mène souvent pour les instructions à une représentation assez différente. Les sources de difficulté sont la recherche statique d'informations auparavant dynamiques, l'absence d'appel de méthode et un indéterminisme possible dans le flot de contrôle.

Nous illustrerons ces changements significatifs avec la représentation des instructions `new`, `if`, `getfield` et `invokevirtual` dont les versions défensives ont déjà été données.

Sémantique de new

Il n'est pas nécessaire pour l'instruction new de la machine virtuelle abstraite de créer l'objet dans le tas, car celui-ci a disparu. Seul reste à pousser dans la pile d'opérande, après avoir vérifié que l'index de classe est une référence valide (test utile pour assurer la « validation croisée » des machines virtuelles (cf section 6.3.2), le type des instances non initialisées à la classe donnée.

```
Definition NEW := [idx:cap_class_idx][state:jcvm_state][cap:jcprogram]
let h = state in
  Cases (Nth_elt (classes cap) idx) of
     None ⇒ (AbortCode class_membership_error state) |
    (Some cl) ⇒
      (update_frame
        (update_opstack (cons (vNonInit idx (p_count h)) (opstack h)) h)
       state)
  end.
```

Sémantique de if_acmp

L'instruction if_acmp pose dans sa version abstraite un problème de déterminisme. En effet, le branchement pour cette instruction s'effectue en fonction des valeurs extraites de la pile. En l'absence de ces valeurs, deux points de branchement sont possibles. Ces deux éventualités (les appels à res_pc2 avec un résultat de la comparaison vrai ou faux) sont collectées sous forme de liste dans le résultat de la fonction.

```
Definition IF_ACMP_COND :=
[oper:opcmp][branch:bytecode_idx][state:jcvm_state]
let h = state in
  Cases (head (opstack h)) of
  None ⇒ (cons (AbortCode opstack_error state) (nil ?)) |
  (Some v2) ⇒
    Cases (head (tail (opstack h))) of
    None ⇒ (cons (AbortCode opstack_error state) (nil ?)) |
    (Some v1) ⇒
      Cases v1 v2 of
      (vRef _) (vRef _) ⇒
        (cons (update_frame (res_pc2 true branch h) state)
          (cons (update_frame (res_pc2 false branch h) state)
        (nil ?))) |
      _        _         ⇒
        (cons (AbortCode type_error state) (nil ?))
  end end end.
```

On remarque donc que l'instruction IF_ACMP_COND possède un type de retour (une liste d'états de retour) différent des instructions vues jusqu'alors. Il en est de même pour les autres instructions de branchement.

Sémantique de getfield

La version abstraite de `getfield` doit rechercher le type statique d'un champ particulier d'une instance de classe dont on ne connait ici que le type statique. Cependant pour toutes les sous-classes d'une classe donnée, le type statique d'un champ reste identique, et l'information sur le type du champ peut être retrouvée à partir du descripteur de classe.

Comparée à la version défensive de la page 59, la version abstraite de `GETFIELD` ne lance plus d'exception. Elle fait appel à la fonction `getfield_obj` en lui donnant comme argument la classe où chercher le champ. Si le type statique fait référence à une instance, la classe est déjà précisée. Sinon, si le type statique est un type de référence, on utilise la classe `java.lang.Object`, super-classe commune de tous les types de référence.

```
Definition GETFIELD := [t:type][idx:instance_field_idx]
[state:jcvm_state][cap:jcprogram]
let h = state in
  Cases (head (opstack h)) of
  None     ⇒ (AbortCode opstack_error state) |
  (Some x) ⇒
    Cases x of
    (vRef (vRef_instance cidx)) ⇒
      (getfield_obj cidx t idx ops state cap) |
    (vRef _) ⇒
      (getfield_obj java_lang_Object t idx ops state cap) |
    (vNonInit _ _) ⇒
      (AbortCode init_error state) |
    _ ⇒ (AbortCode type_error state)
  end end.
```

À partir de cette indication de classe, on recherche le type statique du champ voulu. Comme pour la version défensive, on confronte le type du champ au type avec lequel l'instruction a été appelée.

```
Definition getfield_obj := [cidx:cap_class_idx][t:type][idx:instance_field_idx]
[ops:(list valu)][state:jcvm_state][cap:jcprogram]
Cases (Nth_elt (classes cap) cidx) of
None ⇒ (AbortCap class_membership_error state) |
(Some cl) ⇒
  Cases (Nth_elt (class_var cl) idx) of
  None ⇒ (AbortCap field_error state) |
  (Some nod) ⇒
    Cases nod t of
    (vRef _)         (Ref _)        ⇒ (res_getfield ops nod state) |
    (vPrim vByte)    (Prim Byte)    ⇒ (res_getfield ops (vPrim vShort) state) |
    (vPrim vBoolean) (Prim Byte)    ⇒ (res_getfield ops (vPrim vShort) state) |
    (vPrim vShort)   (Prim Short)   ⇒ (res_getfield ops nod state) |
    (vPrim vInt)     (Prim Int)     ⇒ (res_getfield ops nod state) |
    _                _              ⇒ (AbortCode type_error state)
```

```
end end end.
```

Encore une fois, on ne retrouve pas dans la fonction `res_getfield` de test pouvant conduire à lever une exception (une exception de sécurité pour la version défensive), et la fonction `res_getfield` se limite à construire le résultat.

```
Definition res_getfield :=
[ops:(list valu)][nod:valu][state:jcvm_state]
let h = state in
  (update_frame (push_opstack nod ops h) state).
```

Sémantique de invokevirtual

L'instruction `invokevirtual` conjugue pour sa version abstraite deux spécificités. D'une part, la résolution de la méthode à appeler s'effectue normalement sur un type dynamique absent ici. D'autre part, il n'y a pas création de nouveau contexte d'exécution de méthode, on se contente de pousser le type de retour de la méthode appelée comme le ferait l'instruction `return` à la fin de l'exécution de cette méthode.

```
Definition INVOKEVIRTUAL :=
[nargs:nat][nm:class_method_idx][state:jcvm_state][cap:jcprogram]
let h = state in
  (* nargs must be greater than zero *)
  Cases nargs of
  O     ⇒ (AbortCode args_error state) |
  (S _) ⇒

    Cases (Nth_func (opstack state) nargs) of
    None ⇒ (AbortCode opstack_error state) |
    (Some x) ⇒

      (* Tests if this element is a reference *)
      Cases x of
      (vRef r) ⇒

        (* Get the corresponding class *)
        Cases (Nth_elt (classes cap) (get_type_ref_class_idx r)) of
        None ⇒ (AbortCap class_membership_error state) |
        (Some c) ⇒

          Cases (get_method c nm cap) of
          None ⇒ (AbortCap methods_membership_error state) |
          (Some met) ⇒

            (* Extraction of the list of arguments *)
            Cases (l_take nargs (opstack h)) (l_drop nargs (opstack h)) of
            (Some l) (Some l') ⇒
              (Normal (Build_frame (app_return_type l' (Snd (signature met)))
```

```
                                    (locvars h)
                                    (method_loc h)
                                    (S (p_count h))
                                    (max_opstack_size h))) |
             _ _ ⇒ (AbortCode opstack_error state)
             end |
         end end
      _ ⇒ (inl ? ? (AbortCode type_error state))
end end end.
```

La fonction INVOKEVIRTUAL commence par effectuer les mêmes tests que sa réciproque défensive. Elle extrait ensuite avec la fonction get_type_ref_class_idx l'index de classe du type de référence (java.lang.Object s'il ne s'agit pas d'une instance) et récupère la méthode à l'emplacement spécifié dans les opérandes du code octet. L'état final est contruit en enlevant de la pile d'opérande les arguments de la méthode et en y rajoutant avec la fonction app_return_type le type de retour de la méthode appelée. Le pointeur de programme est incrémenté (il ne l'est dans la version défensive que lors de l'exécution du bytecode return de la méthode appelée).

4.4 Jakarta

Jakarta est un logiciel dont le but est de fournir un environnement interactif pour la transformation de machines virtuelles et pour l'aide à la vérification dans un assistant de preuve de la correction des ces transformations. Des exemples de transformations sur les machines virtuelles sont donnés par le raffinement, l'optimisation, mais nos efforts ont porté principalement sur les abstractions. Pour une description plus complète de *Jakarta*, se rapporter à la thèse de Simão MELO DE SOUSA [92] ou bien encore à [10, 12, 13].

Dans notre optique, *Jakarta* sera le support de la construction automatique et la validation croisée de machines virtuelles selon une propriété arbitraire P. Concrètement, nous fournissons à *Jakarta* les entrées suivantes :

– une machine virtuelle P-défensive manipulant des valeurs typées (dont la notion est déterminée par P) et assurant P à l'exécution par un mécanisme de vérification de type;
– une fonction d'abstraction entre les types de la machine virtuelle P-défensive et ceux de la machine virtuelle que l'on souhaite obtenir (par exemple machine virtuelle offensive ou abstraite) ;
– un script de transformation guidant l'abstraction dans les cas où la définition de la fonction d'abstraction n'est pas suffisante

et nous obtenons :

- une machine virtuelle abstraite selon la fonction d'abstraction ;
- les preuves que cette dernière constitue une transformation correcte de la machine virtuelle P-défensive.

Nous présenterons dans cette partie les mécanismes d'abstraction et nous aborderons dans le chapitre 6 la construction des preuves de correction.

4.4.1 Le langage de spécification

Le cœur de *Jakarta* est constitué par un langage de spécification (JSL pour Jakarta Specification Language) décrivant les machines virtuelles dans un style mathématique neutre. JSL est un langage à types polymorphes dont l'exécution est basée sur la réécriture de termes [4, 23]. Ce langage se veut relativement simple afin de faciliter au maximum le lien avec les assistants de preuves ou les langages de programmation.

Les expressions Les expressions du langages JSL sont des termes du premier ordre construits à partir de variables et de symboles de constantes. Ces derniers sont ou bien des symboles de constructeur, introduits par une déclaration de type de données, ou bien des symboles introduits par une définition de fonction. JSL propose également une syntaxe concrète pour les enregistrements (record), cependant ceux-là sont traduits, comme le fait Coq, en un type inductif à un seul constructeur et n'apparaissent alors pas dans les expressions du langage.
De manière formelle, les expressions sont définies comme suit :

Définition 1 (Expressions) *Soient $\mathcal{C}$ un ensemble de symboles de constructeurs, $\mathcal{D}$ un ensemble de symboles de fonctions et $\mathcal{V}$ un ensemble de variables, alors l'ensemble $\mathcal{E}$ des expressions JSL est donné par la syntaxe suivante :*

$$\mathcal{E} ::= \mathcal{V} \mid \mathcal{C}\ \vec{\mathcal{E}} \mid \mathcal{D}\ \vec{\mathcal{E}}$$

où il est requis que les arités des éléments de $\mathcal{C}$ et $\mathcal{D}$ soient respectées.

Les règles Les fonctions sont représentées en *Jakarta* par l'intermédiaire de règles de réécriture. Dans le cadre de l'abstraction et par rapport la représentation des langages fonctionnels avec filtrage de motifs, cela présente les avantages suivants :

- disposer directement de toutes les valeurs pouvant être prises par une fonction suivant les filtrages réalisés, pour être alors en mesure de travailler indépendamment sur chacune d'entre elles ;

– obtenir plus de souplesse dans le structure des fonctions manipulées autorisant ainsi par exemple l'écriture de fonctions non-déterministes.

Néanmoins, la traduction vers et depuis un langage à filtrage de motifs reste présente, en vue de la réutilisation des formalisations existantes.

Le lien avec le filtrage de motifs est rendu encore plus fort par les conditions de construction des règles de réécriture et la réutilisation de la notion de motif.

Définition 2 (Motifs) *L'ensemble $\mathcal{P}$ des motifs est le sous-ensemble de $\mathcal{E}$ défini par la syntaxe :*

$$\mathcal{P} ::= \mathcal{V} \mid \mathcal{C}\ \vec{\mathcal{P}}$$

où dans la seconde partie, les motifs doivent avoir un ensemble de variables disjointes.

Ces motifs sont réutilisés dans les membres droits des règles de réécriture, à la manière du filtrage.

Définition 3 (Règles de réécriture) *Les règles de réécriture des fonctions du langage JSL sont de la forme :*

$$l_1 \twoheadrightarrow r_1,\ \ldots\ , l_n \twoheadrightarrow r_n \Rightarrow g \rightarrow d$$

où :

- $\vec{r_i} \in \mathcal{P}$, $\vec{l_i}, g, d \in \mathcal{E}$, *et* $g = f\ \vec{x}$ *(avec* $\vec{x} \in \mathcal{V}$*) sont deux à deux distincts ;*
- $\mathsf{var}(l_k) \subseteq \mathsf{var}(g) \cup \mathsf{var}(r_1) \cup\ \ldots\ \cup \mathsf{var}(r_{k-1})$ *et* $\mathsf{var}(d) \subseteq \mathsf{var}(g) \cup \mathsf{var}(\vec{r_i})$*;*
- $\mathsf{var}(r_k) \cap \mathsf{var}(g) = \emptyset$ *et* $\mathsf{var}(r_j) \cap \mathsf{var}(r_k) = \emptyset$ *si* $j \neq k$*.*

Ces contraintes rendent ainsi la liaison avec le filtrage de motifs évidente, par exemple, les règles suivantes :

$$\begin{array}{lcl} l \twoheadrightarrow Nil & \Rightarrow & (is_empty\ l) \rightarrow True \\ l \twoheadrightarrow (Cons\ el\ tl) & \Rightarrow & (is_empty\ l) \rightarrow False \end{array}$$

correspondent sous forme arborescente aux deux branches de la fonction ML suivante :

```
let is_empty l = match l with [] → True | el::tl → False
```

Les types Par dessus le langage de spécification précédemment décrit, vient se greffer un système de types, clé de la conduite de la génération d'abstraction.

Définition 4 (Types) *Soient les ensembles $\mathcal{V}_\mathcal{T}$ de variables de type, $\mathcal{T}_d$ de symboles de types de données, $\mathcal{T}_a$ de symboles de types abstraits et $\mathcal{T}_s$ de symboles de types synonymes, alors l'ensemble $\mathcal{T}$ des types JSL est donné par la syntaxe suivante :*

$$\mathcal{T} ::= \mathcal{V}_\mathcal{T} \mid \mathcal{T}_d\ \mathcal{T}^* \mid \mathcal{T}_a\ \mathcal{T}^* \mid \mathcal{T}_s\ \mathcal{T}^*$$

où il est requis que les arités des éléments de $\mathcal{T}_d$, $\mathcal{T}_a$ et $\mathcal{T}_s$ soient respectées.

Aux symboles de constructeurs ou de fonctions sont associés des schémas de types, c'est-à-dire des expressions closes de la forme :

$$\forall \alpha_1 \ldots \alpha_m.\ \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \tau$$

où $\alpha_1, \ldots, \alpha_m \in \mathcal{V}_\mathcal{T}$ et $\sigma_1, \ldots, \sigma_n, \tau \in \mathcal{T}$.

On notera alors $\mathcal{E}_\mathcal{T}$ l'ensemble des expressions de type, constitué de l'ensemble des types et des schémas de type.

Le type `valu` des valeurs de la machine virtuelle défensive s'écrit par exemple en *Jakarta* sous la forme :

```
data valu  =
        VPrim valu_prim |
        VRef valu_ref
and
data valu_ref  =
        VRef_null |
        VRef_array type0 heap_idx |
        VRef_instance cap_class_idx heap_idx |
        VRef_interface cap_interf_idx heap_idx.
```

De manière usuelle, le type des expressions est vérifié (par un vérificateur de type décrit dans [92]) relativement à un contexte qui associe un type à chaque variable. En raison des synonymes de type, la vérification de type est faite modulo la relation sur ces synonymes de type.

Les fonctions Il est maintenant possible de définir les fonctions du langage JSL :

Définition 5 (Fonctions) *Une fonction JSL f est définie par la donnée de son type* `f:` $\mathtt{T}_1 \rightarrow \ldots \rightarrow \mathtt{T}_m \rightarrow \mathtt{U}$ *et d'un ensemble de règles de réécriture $l_1 \twoheadrightarrow r_1, \ \ldots\ , l_n \twoheadrightarrow r_n \Rightarrow g \rightarrow d$ dont le symbole de tête de g est f.*

La fonction IF_ACMP_COND de la page 58 est traduite en *Jakarta* sous la forme suivante :

```
function iF_ACMP_COND  :
   opcmp→bytecode_idx→jcvm_state→returned_state :=

<iF_ACMP_COND_rule_1>
(stack_f state)→Nil
   ⇒ (iF_ACMP_COND oper branch state)→
        (abortCode State_error state);

<iF_ACMP_COND_rule_2>
(stack_f state)→(Cons h lf),
(head (opstack h))→(Value v2),
(head (tail (opstack h)))→(Value v1),
v1→(VPrim v)
   ⇒ (iF_ACMP_COND oper branch state)→
        (abortCode Type_error state);

<iF_ACMP_COND_rule_3>
(stack_f state)→(Cons h lf),
(head (opstack h))→(Value v2),
(head (tail (opstack h)))→(Value v1),
v1→(VRef vx),
v2→(VPrim v)
   ⇒ (iF_ACMP_COND oper branch state)→
        (abortCode Type_error state);

<iF_ACMP_COND_rule_4>
(stack_f state)→(Cons h lf),
(head (opstack h))→(Value v2),
(head (tail (opstack h)))→(Value v1),
v1→(VRef vx),
v2→(VRef vy)
   ⇒ (iF_ACMP_COND oper branch state)→
        (update_frame (res_pc2_nat
          (res_acompare2 oper (vr2hi vx) (vr2hi vy))
         branch h) state);

<iF_ACMP_COND_rule_5>
(stack_f state)→(Cons h lf),
(head (opstack h))→(Value v2),
(head (tail (opstack h)))→Error
   ⇒ (iF_ACMP_COND oper branch state)→
        (abortCode Opstack_error state);

<iF_ACMP_COND_rule_6>
(stack_f state)→(Cons h lf),
(head (opstack h))→Error
   ⇒ (iF_ACMP_COND oper branch state)→
        (abortCode Opstack_error state)
.
```

On y observe clairement les différents chemins d'exécution de la fonction.

Il est possible de définir un modèle d'exécution sur ces fonctions. En reprenant les notations de [20] :

Définition 6 (Modèle d'exécution) *Soit R un ensemble de règles de réécriture. Une expression s se réécrit en t par R, que l'on note $s \rightarrow_R t$, s'il existe une règle r de R*

$$l_1 \twoheadrightarrow r_1, \ \ldots \ , l_n \twoheadrightarrow r_n \Rightarrow g \rightarrow d$$

une position p dans s et une substitution θ des variables de R telles que

- $s_{|_p} = \theta g$ *et* $t = s[p \leftarrow \theta d]$*;*
- *pour* $1 \leq i \leq n$, $\theta l_i \rightarrow_R^* \theta r_i$*;*

où $\rightarrow_R^*$ est la clôture réflexive et transitive de $\rightarrow_R$.

On notera qu'aucune contrainte n'assure l'exhaustivité ou la confluence des règles de réécriture définissant une fonction. On se permet ainsi de définir dans le modèle JSL des fonctions partielles ou non-déterministes.

4.4.2 Génération d'abstractions

Dans cette section, nous montrerons en quoi la forme linéaire des règles de réécriture par lesquelles sont définies les fonctions dans JSL aide à la manipulation, et plus spécifiquement à l'abstraction, des spécifications.

Dans *Jakarta*, la génération d'abstraction, à partir d'une spécification `S` se déroule comme suit :

1. l'utilisateur fournit pour chaque type de donnée τ qu'il souhaite abstraire dans `S` le type abstrait $\hat{\tau}$ correspondant ainsi que la fonction d'abstraction α_τ de τ vers $\hat{\tau}$ en langage JSL ;
2. l'utilisateur construit un script d'abstraction à partir des données précédentes et d'informations sur le comportement de l'abstraction dans les cas où les procédures automatiques sont imprécises ;
3. le moteur de transformation de *Jakarta* génère automatiquement pour chaque fonction $\texttt{f: } \texttt{T}_1 \rightarrow \ldots \rightarrow \texttt{T}_n \rightarrow \texttt{U}$ de `S` sa contrepartie abstraite $\hat{\texttt{f}}\texttt{:}\hat{\texttt{T}}_1 \rightarrow \ldots \rightarrow \hat{\texttt{T}}_n \rightarrow \hat{\texttt{U}}$ par une traduction syntaxique de ses règles de réécriture. Dans cette dernière phase, d'autres opérations, comme la suppression des conditions inutiles, peuvent avoir lieu.

Dans ce processus d'abstraction, le rôle du script se limite aux endroits où l'intervention de l'utilisateur est nécessaire. La plupart de l'abstraction s'effectue de manière automatique. Par exemple, pour la construction de la machine virtuelle offensive, seules 12 lignes de script sont nécessaires.

Traduction syntaxique

Nous allons présenter dans cette section comment s'effectue la phase de traduction syntaxique d'une spécification JSL à partir de la donnée de fonctions d'abstraction de types.

Fonctions d'abstraction La traduction s'effectue par la donnée de fonctions d'abstraction d'un type vers un autre. Les types de données concrets en JSL s'exprimant par des types inductifs, les fonctions d'abstraction traduisent alors, par l'intermédiaire des règles de réécriture de JSL, un constructeur d'un type vers un constructeur du type abstrait. Plus formellement, une fonction d'abstraction σ d'un type τ vers un type τ_a sera décrite par des règles de la forme :

$$x \twoheadrightarrow r \Rightarrow \sigma x \rightarrow r_a$$

où r et r_a sont des motifs de τ et τ_a respectivement.

Notations Dans la suite, nous désignerons par $\mathcal{A}$ la fonction qui à un type τ retourne la fonction d'abstraction pour ce type (qui peut être l'identité id). Par extension, pour un constructeur c de type τ, nous noterons par $\mathcal{A}(c)$ la fonction d'abstraction qui abstrait les éléments de τ. Finalement, nous désignerons par $\hat{f}$ l'abstraction du symbole f (qui peut être f lui-même s'il n'est pas touché par l'abstraction).

L'abstraction d'une fonction se réalisera par l'abstraction de son type et de chacune de ses règles. La traduction syntaxique agira donc sur les types et sur les expressions que contiennent les règles.

Définition 7 (Abstraction de types) *Soient τ un type de $\mathcal{T}$ et $\sigma : \tau \rightarrow \tau_a$ telle que $\mathcal{A}(\tau) = \sigma$ alors l'opérateur d'abstraction de type, noté $\lceil . \rceil : \mathcal{T} \rightarrow \mathcal{T}$ est défini par $\lceil \tau \rceil = \tau_a$.*

Il est étendu aux schémas de types par $\lceil . \rceil : \mathcal{E}_\mathcal{T} \rightarrow \mathcal{E}_\mathcal{T}$ défini par :

$$\lceil \forall \alpha_1 \ \ldots \alpha_m . \ \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \tau \rceil \leadsto \forall \alpha_1 \ \ldots \alpha_m . \ \lceil \sigma_1 \rceil \rightarrow \ldots \rightarrow \lceil \sigma_n \rceil \rightarrow \lceil \tau \rceil$$

où $\alpha_1, \ldots, \alpha_m \in \mathcal{V}_\mathcal{T}$ et $\sigma_1, \ldots, \sigma_n, \tau \in \mathcal{T}$ et où le symbole $\leadsto$ prend le sens « est traduit en ».

Définition 8 (Abstraction d'expressions) *L'opérateur $\lceil . \rceil : \mathcal{E} \rightarrow \mathcal{E}$ d'abstraction d'expression est récursivement défini par les règles suivantes :*

$$\frac{x \in \mathcal{V}}{\lceil x \rceil \leadsto x} \qquad \frac{f \in \mathcal{D}}{\lceil (f \ t_1 \ldots t_n) \rceil \leadsto \hat{f} \ \lceil t_1 \rceil \ldots \lceil t_n \rceil}$$

$$\frac{c \in \mathcal{C} \qquad \mathcal{A}(c) = \mathrm{id}}{\lceil c\ t_1 \dots t_n \rceil \rightsquigarrow c\ \lceil t_1 \rceil\ \dots\ \lceil t_n \rceil}$$

$$\frac{\begin{array}{c} c \in \mathcal{C} \\ \mathcal{A}(c) = \alpha \wedge\ (x \twoheadrightarrow (c\ x_1 \dots x_n)\ \Rightarrow (\alpha\ x) \rightarrow (c'\ y_1 \dots y_p)) \in \alpha \\ \sigma = [x_1 \leftarrow t_1\ \dots\ x_n \leftarrow t_n] \end{array}}{\lceil c\ t_1 \dots t_n \rceil \rightsquigarrow (c'\ \sigma(y_1)\ \dots\ \sigma(y_p))}$$

L'abstraction d'une variable est la variable elle-même, cependant son type est abstrait, ce qui n'apparaît pas ici. Pour une fonction, on abstrait le symbole de fonction et récursivement les arguments de la fonction. Enfin, pour un motif, si le type auquel appartient le constructeur ne fait pas partie des types à abstraire, on se contente d'abstraire ses arguments. Dans le cas contraire, on applique au motif la substitution, induite par la fonction d'abstraction, lui faisant correspondre sa version abstraite.

Définition 9 (Abstraction de fonction) *L'abstraction d'une fonction f de type τ se réalise en traduisant chaque règle*

$$l_1 \twoheadrightarrow r_1,\ \dots\ , l_n \twoheadrightarrow r_n \Rightarrow g \rightarrow d$$

de f en :

$$\lceil l_1 \rceil \twoheadrightarrow \lceil r_1 \rceil,\ \dots\ , \lceil l_n \rceil \twoheadrightarrow \lceil r_n \rceil \Rightarrow \lceil g \rceil \rightarrow \lceil d \rceil$$

et en affectant à la fonction $\hat{f}$ produite le type $\lceil \tau \rceil$.

En réalité, dans *Jakarta* l'abstraction se présente sous une forme légèrement plus complexe (voir [92] pour plus de détails). Ceci est dû au fait que l'on permet de donner comme fonction d'abstraction une fonction dont les règles peuvent être de la forme :

$$x \twoheadrightarrow r \Rightarrow \alpha x \rightarrow t$$

c'est-à-dire que αx se réécrit en un terme t (appel d'une autre fonction) et pas forcément en un motif.

Notons de plus que dans cette présentation, la présence des types synonymes peut poser problème car il est possible d'écrire des fonctions d'abstractions différentes pour des types synonymes d'un même type concret.

Enfin, la traduction syntaxique peut conduire à des fonctions qui contiennent des règles inutiles, qui présentent des formes de non-déterminisme, ou qui deviennent partielles. Il est cependant possible de les transformer encore pour y remédier.

Transformations des fonctions traduites

À l'issue de la phase de traduction syntaxique, il est fréquent que l'on obtienne pour une fonction des règles identiques, une seule d'entre elles est alors conservée.

Un autre problème plus complexe survient lorsqu'on obtient pour une même fonction deux règles dont les conditions sont identiques (éventuellement à une substitution des variables près) mais les conclusions différentes, c'est un cas de non-déterminisme. Pour y remédier, on peut soit :

- conserver les conditions et collecter tous les résultats possibles sous forme de liste. Il faut alors changer le type de la fonction et modifier toutes les autres fonctions qui l'appellent pour refléter ce changement.
- rajouter des procédures de décision dans le script d'abstraction (voir section 4.4.2) pour éliminer les cas non souhaités.

Enfin, une fonction peut devenir partielle c'est-à-dire, par comparaison au filtrage de motif, ne pas proposer de résultat pour tous les constructeurs du type filtré. Il est alors possible de rendre totale la fonction en élevant le type et en rendant une valeur par défaut, le joker des langages à filtrage de motif, pour les autres cas. Ceci est réalisé par le type `option`, de constructeurs `None` (la valeur par défaut) et `Some` α où α est une variable représentant le type élevé. Il faut ici, comme dans le cas du non-déterminisme, modifier également les fonctions appelant la fonction rendue totale pour qu'elles prennent en compte ce changement.

Ouverture du langage de spécification

Le langage de spécification JSL présenté à la section 4.4.1 reste spécifique à *Jakarta*. Il est important de prévoir des passerelles vers des langages de programmation ou des assistants de preuve afin de pouvoir récupérer des spécifications existantes ou d'utiliser les abstractions produites.

Jakarta propose alors une forme alternative de présentation des spécifications permettant une traduction aisée vers les langages supportant le filtrage de motifs et les définitions récursives (on recouvre alors de très nombreux langages de programmation comme OCAML ou Scheme ainsi que des assistants de preuve comme COQ ou Isabelle). Ce format de représentation alternative des données est appelé JIR, pour Jakarta Intermediate Representation.

Définition 10 (Expressions JIR) *L'ensemble* $\mathcal{E}_c$ *des expressions JIR est*

défini par la syntaxe suivante :

$$\mathcal{E}_c ::= \mathcal{V} \mid \ \mathcal{C}\ \vec{\mathcal{E}_c} \mid \ \mathcal{D}\ \vec{\mathcal{E}_c} \mid \quad \text{case } \mathcal{E}_c \text{ of } \{\mathcal{M}\}$$

où l'ensemble $\mathcal{M}$ des motifs est défini par :

$$\mathcal{M} ::= \{\} \quad \mid \quad \mathcal{M} \mid \mathcal{P} \Rightarrow \mathcal{E}_c$$

Les types JIR restent identiques aux types JSL.

JIR vers JSL La phase de traduction de JIR vers JSL consiste à passer d'une représentation arborescente des filtrages effectués par une fonction à une représentation linéaire. Ce sens de traduction ne présente pas de difficultés particulière car le format JSL offre plus d'expressivité que le format JIR. Du point de vue algorithmique, il suffit d'énumérer tous les chemins de la racine aux feuilles de la représentation arborescente de la fonction.

JSL vers JIR Ce sens de traduction n'est possible que lorsque les fonctions JSL respectent des critères de bonne formation (déterminisme et exhaustivité du filtrage). Du point de vue algorithmique, il faut cette fois pouvoir faire correspondre à une position des règles de réécriture JSL une position dans l'arbre JIR et rechercher à cette position dans les autres règles de réécriture de la fonction les motifs filtrés.

De et vers JIR Les traductions entre JIR et JSL mises en place, il faut encore réaliser les traductions entre JIR et le système voulu. Passer de la syntaxe du système voulu à JIR est le sens le plus complexe puisqu'il faut être capable d'accéder à la syntaxe abstraite du système pour le traduire vers JIR. En revanche, le sens inverse est généralement plutôt aisé puisqu'il consiste simplement à afficher dans la syntaxe du système cible la syntaxe abstraite de JIR (en utilisant les mots-clés du langage cible, comme `let` ou `match` pour OCAML).

Traductions proposées Actuellement les traductions vers OCAML, COQ, Isabelle, et PVS par l'intermédiaire de JIR sont disponibles. Elles ont permis par exemple de récupérer dans *Jakarta* l'intégralité des spécifications en COQ de Certicartes.
Notons aussi qu'une traduction vers Spike [26] est possible, elle est effectuée directement à partir de JSL en raison de l'utilisation commune des règles de réécriture.

Les scripts d'abstraction

Le script d'abstraction permet l'interaction entre l'utilisateur et le système *Jakarta*. On a déjà vu qu'il devait contenir les fonctions d'abstraction. On devine qu'il doit préciser également la spécification à abstraire. La dernière partie du script contient les directives à suivre lors de l'abstraction pour guider cette dernière.

Pour illustrer l'utilité de ces directives, nous allons nous placer dans le cadre de l'abstraction de type afin d'obtenir une machine offensive. Une fonction d'abstraction précise comment changer le type `valu` des valeurs en le type `z`. Pour les fonctions vérifiant le type des valeurs, la version défensive de la machine opère un filtrage sur une valeur `v` de type `valu`, par exemple :

```
Cases v of
  (vInt z) ⇒ ...
| (vShort z) ⇒ (AbortCode Type_error state)
| ...
end
```

qui s'écrit en *Jakarta* par des règles successives :

```
..., v → (vInt z), ... ⇒ g → ...
..., v → (vShort z), ... ⇒ g → (AbortCode Type_error state)
```

À la fin de la traduction, on se trouvera avec des règles à prémisses communes, mais à conclusions différentes :

```
..., v → z, ... ⇒ g → ...
..., v → z, ... ⇒ g → (AbortCode Type_error state)
```

Puisque l'on souhaite ne plus conserver les états menant à une erreur de typage dans la machine offensive, une commande (`reject`) permet de supprimer toutes les règles dont la conclusion est une instance de l'argument de la commande. Pour suivre notre exemple, nous préciserons de rejeter toutes les règles dont la conclusion est `(AbortCode Type_error state)` qui identifie dans notre formalisation les états résultant d'une erreur de typage. Nous levons ainsi aussi une source de non-déterminisme.

Parmi les autres directives disponibles, citons :

- la commande `replace` substituant l'appel d'une fonction par une autre, utile par exemple lors de la construction de la machine abstraite, remplace une fonction de recherche dynamique (dans le tas) par une fonction de recherche statique (à partir des types) ;
- la commande `drop` permettant d'enlever un argument devenu inutile d'une fonction, par exemple un argument calculé à partir d'informations disparaissant lors de l'abstraction ;

– les commandes `determine` et `totalize` qui rendent respectivement déterministe et totale une fonction signalée comme ne l'étant pas après une première tentative d'abstraction.

4.5 Conclusion

Dans cette partie, nous avons décrit la construction à l'intérieur même de Coq d'abstractions de machine virtuelle défensive (de la partie précédente) vis-à-vis de la propriété de sûreté du typage. Les machines virtuelles réalisées, offensive et abstraite, serviront respectivement pour une exécution efficace des programmes et pour la vérification statique de la sûreté du typage de programmes, par l'intermédiaire d'un vérificateur de code octet.

Nous dégageons ensuite de ce travail une méthodologie générale visant à assurer des propriétés de sécurité sur la machine virtuelle par la construction d'une machine virtuelle défensive et la dérivation à partir de cette dernière de machines virtuelles offensive et abstraite et d'un vérificateur de code octet. Ces propriétés, outre la sûreté du typage peuvent concerner le contrôle des ressources, l'initialisation des objets.

Nous avons ensuite présenté l'outil *Jakarta* proposant un procédé automatique pour la construction des machines virtuelles offensive et abstraite (et également l'aide à la preuve de propriétés nécessaires au vérificateur de code octet, voir chapitre 6). Cet outil a été testé avec notre machine virtuelle défensive et a généré les machines virtuelles offensive et abstraite, en nécessitant un script d'abstraction de seulement une dizaine et de moins d'une centaine de lignes respectivement (nécessité de réécrire certaines fonctions). Les machines virtuelles générées (plus de 5000 lignes de code chacune) sont parfaitement semblables à celles réalisées manuellement dans Coq validant alors l'approche suivie.

CHAPITRE 5

Vérification de bytecode

La vérification de code octet est un processus essentiel dans le dispositif de sécurité de *Java Card* (et plus largement *Java*). Il repose sur une analyse statique approfondie du programme dont le but est d'assurer un comportement correct du programme lors de son exécution.

Durant les dernières années, de nombreux projets ont été réalisés dans le but de prouver formellement la correction du vérificateur de code octet. Cependant, peu d'efforts ont été fournis afin de fournir des techniques, méthodologies et outils pour aider à de telles formalisations.

Nous présentons ici une méthodologie qui factorise la formalisation en :

- un composant spécifique, qui traduit la sémantique opérationnelle de machines virtuelles défensive et abstraite ;
- un composant générique, qui conduit à la construction de vérificateur de code octet à partir d'une machine virtuelle (en utilisant une analyse de flot de données pour aboutir à un calcul de point fixe) et à l'utilisation de techniques compositionnelles pour procéder à cette vérification méthode par méthode.

Cette méthodologie repose presque entièrement sur la construction préalable d'une machine virtuelle défensive à partir de laquelle on dérive les machines virtuelles abstraites et offensives par des techniques d'abstraction, et ce de manière presque automatique (voir section 4.4).

La correction de la vérification de code octet s'exprime alors comme une

conséquence de la correction :

- des abstractions de la machine virtuelle défensive pour la partie spécifique aux propriétés à assurer (validation croisée des machines virtuelles) ;
- la dérivation du vérificateur de code octet à partir de la machine virtuelle abstraite. La preuve de correction de cette dérivation est générique et s'applique à d'autres propriétés que le typage, comme par exemple le contrôle du flot d'information ou des ressources.

Nous fournissons pour cela une interface permettant de construire un vérificateur de code octet à partir de la validation croisée des machines virtuelles. Cette interface est plus simple à instancier que les autres cadres formels visant à justifier la vérification de code octet et facilite la tâche de fournir la preuve de correction de la partie spécifique.

En dehors de cette interface, qui constitue une contribution principale, nous apportons un nombre intéressant, bien que plus ciblés, de résultats :

- une justification formelle des techniques de vérification de code octet compositionnelles. Plus précisément, nous apportons la preuve dans un cadre abstrait que la vérification méthode par méthode garantit que les programmes acceptés se comporteront correctement. Nous prenons en compte également la gestion des exceptions.
- une preuve de correction formelle d'un vérificateur de code octet paramétrisé (voir section 5.3.2) pouvant être instancié pour effectuer une analyse monovariante, polyvariante ou polyvariante avec optimisations suivant le compromis choisi entre efficacité et précision.
- une spécification concise d'un vérificateur léger (et paramétré) de code octet, adapté aux dispositifs aux ressources limitées. Une telle spécification n'est pas nouvelle, mais notre développement formel court et abstrait permet un raffinement facile de la notion de certificat (voir section 5.1.4).

Enfin, notre formalisation fournit un intéressant cas d'étude pour le système de modules de Coq, nouvellement introduit avec la version 7.4.

5.1 Aperçu de la vérification de bytecode

Nous allons donner dans cette section un aperçu de la vérification de code octet. Pour une revue plus détaillée des algorithmes et formalisations, nous invitons le lecteur à se reporter aux articles de Leroy [84] et de Hartel et Moreau [63].

La vérification de code octet en *Java Card* comporte deux phases distinctes. La première des ces phases consiste en une analyse structurelle du fichier *capfile* (ce que réalise pour nous le *JCVM Tools*). Les vérifications à effectuer sont détaillées dans [121, Chapitre 6] ou [126] et portent sur les points suivants :

– cohérence globale du *capfile* (nombre, type et taille des composants) ;
– utilisation correcte du mot-clé `final` pour les méthodes et les classes ;
– présence d'une super-classe pour toutes les classes (sauf `Object`) ;
– cohérence de la zone des constantes (références valides).

Cette analyse, bien qu'éventuellement coûteuse en nombre de tests à effectuer, reste sur le principe simple à effectuer.

La seconde phase est la plus complexe et la plus cruciale du point de vue de la sécurité. Elle vise à :

– vérifier l'utilisation des valeurs en accord avec leur type (cela évite par exemple l'emploi opérations arithmétiques sur les valeurs de référence qui donnerait alors accès à des zones de mémoire interdites) ;
– détecter les débordements de pile (et ainsi l'écrasement possible de certaines zones de mémoire) et les piles vides pour les instructions nécessitant de dépiler des valeurs ;
– détecter les pointeurs de programmes invalides (en dehors du code de la méthode) ;
– assurer l'utilisation des méthodes dans un contexte compatible avec leur visibilité (`public`, `protected`, `private`) ;
– assurer l'initialisation des registres (l'accès en lecture à un registre doit s'effectuer après un accès en écriture à ce registre) ;
– assurer l'initialisation correcte des objets (lors de la création d'une instance de classe, le constructeur de la classe créée, ainsi que ceux de toutes ses super-classes, doit être appelé avant toute utilisation de l'instance).

On remarquera que tous ces tests sont effectués par la machine défensive, mais l'utilisation préalable d'un vérificateur de code octet permet de se dispenser une fois pour toutes de ces tests lors de l'exécution. On pourra ainsi utiliser pour l'exécution, et avec la même sécurité, la machine offensive, plus rapide.

5.1.1 Algorithme

La partie la plus complexe du processus de vérification de code octet, la vérification du bon typage des valeurs, s'effectue par un algorithme d'ana-

lyse de flot de données. La plupart des vérificateurs de code octet utilisent l'*algorithme de* KILDALL [75], un algorithme générique d'analyse du flot de programme reposant sur l'existance d'une *fonction d'optimisation* (la fonction d'unification dans ce qui sera décrit ci-dessous, notion proche des concepts d'élargissement et de rétrécissement de l'interprétation abstraite [49]). La description d'un vérificateur de code octet pour *Java* dans [126] ou [85, section 4.9.2] reprend et adapte l'algorithme de KILDALL. Cependant, il existe d'autres algorithmes, comme ceux présentés par QIAN basés sur une itération de point fixe chaotique [102] ou par COGLIO et ses co-auteurs [43] décrit en Specware [72] et basé sur la génération de contraintes.

Nous allons donner une vue d'ensemble de l'algorithme de KILDALL appliqué au cas de *Java*.

Le vérificateur procède méthode par méthode. Il se sert de la machine virtuelle abstraite et d'un historique des états contenant pour chaque instruction le dernier état obtenu avant exécution de cette instruction ainsi qu'un booléen indiquant si un changement est intervenu. Les booléens de cet historique sont initialisés à faux sauf pour la première instruction. Enfin, l'historique pour la première instruction reçoit l'état normalement présent lors du début de l'exécution de la méthode analysée (pile d'opérande vide et variables locales contenant les arguments de la méthode).

L'algorithme se déroule comme suit, jusqu'à épuisement des booléens marqués à vrai dans l'historique :

1. Choisir une instruction de l'historique pour lequel le booléen est marqué à vrai. Changer la valeur du booléen.
2. Exécuter la machine virtuelle abstraite sur l'état de l'historique choisi à l'étape 1.
3. Déterminer tous les successeurs possibles après exécution (il s'agit pour la plupart des instructions, si l'exécution s'est bien passée, de l'instruction suivante). Pour chacun des successeurs :
   - S'il est atteint pour la première fois, placer dans l'historique l'état résultant de l'exécution et mettre le booléen pour cette instruction à vrai ;
   - Sinon, unifier (voir plus bas) l'état résultant de l'exécution avec l'état sauvegardé dans l'historique. Si l'unification échoue alors la vérification termine et échoue également. Dans le cas contraire, si l'état résultant de l'unification est différent de l'état de l'historique, le placer dans l'historique et mettre le booléen à vrai.
4. Revenir à l'étape 1.

La vérification de la méthode réussit donc lorsque tous les booléens de l'historique sont à faux et que l'unification n'a jamais échoué.

Pour pouvoir unifier deux états, ces états doivent avoir le même nombre de variables locales et d'éléments dans leur pile d'opérande. Pour les deux états, les éléments (dont on rappelle qu'ils représentent des types) des variables locales et de la pile doivent pouvoir ensuite s'unifier entre eux. Si ces éléments sont différents et qu'ils ne représentent pas des types de référence, l'unification échoue. S'ils représentent des types de référence, le résultat de l'unification est le type commun, dans la relation d'héritage entre types de référence, de ces deux types.

L'unification décrite ici fonctionne correctement dans les programmes ne contenant pas de sous-routines. Elle devient cependant problématique autrement, en n'acceptant pas certains programmes corrects du point de vue du typage.

5.1.2 Le problème des sous-routines

Une sous-routine est le nom donné à un bloc d'instructions pouvant être appelé de divers points à l'intérieur d'une méthode. Elle peut-être vue comme un méthode ne nécessitant pas de création de contexte d'exécution. Bien qu'elle permette une exécution rapide (par rapport à l'appel d'une méthode) de code partagé par plusieurs points de programme de la méthode, elle est principalement utilisée pour implémenter la construction `try-finally` du langage *Java* et éviter la duplication de code (on consultera [56] pour une idée des coûts et bénéfices des sous-routines).

La machine virtuelle *Java* (et par analogie *Java Card*) gère les sous-routines grâce aux instructions `jsr` et `ret`. La première de ces instructions pousse une adresse de retour (`ReturnAddress`) dans la pile et place le compteur de programme à l'emplacement indiqué comme opérande ; la seconde place le compteur de programme au point indiqué par l'adresse de retour récupérée dans les variables locales.

Les sous-routines compliquent singulièrement l'analyse de flot de données du vérificateur de code octet. En effet, d'une part les successeurs possibles de l'instruction `ret` dépendent non pas de l'instruction seule mais des valeurs contenues dans l'état (la valeur du registre contenant l'adresse de retour) et d'autre part, pouvant être appelée de différents emplacements du programme, la phase d'unification à l'entrée de la sous-routine peut conduire à une perte de précision trop importante dans les types des variables locales.

Le problème est illustré en priorité par l'unification *a priori* non prévue d'adresses de retour différentes, comme cela se produit à l'entrée d'une sous-routine appelée de plusieurs emplacements du programme. Mais le problème se pose tout autant par l'unification des variables locales non-utilisées dans la sous-routine.

Nous alloms suivre l'exemple de LEROY dans [84] pour le programme de la figure 5.1. On observe sur cette figure qu'à l'entrée de la sous-routine à la ligne 20, il faut, selon l'algorithme décrit précédemment, unifier les états résultant des deux passes sur cette instruction. Or d'une part, nous serons conduits à unifier deux adresses de retour différentes et d'autre part le résultat de l'unification en un registre non initialisé $\top$ et `int` ne pouvant être que $\top$ (pour le registre 0), la propagation de ce résultat à l'instruction 13 provoquera une erreur. Pourtant comme le registre 0 n'est pas utilisé dans la sous-routine, son type ne devrait pas changer entre l'entrée et la sortie de la sous-routine.

Pouvant considérer également que les sous-routines peuvent être répliquées *en ligne* à l'endroit où elles sont appelées sans changer la sémantique du programme, et qu'alors la vérification aurait réussi, rien ne justifie alors d'unifier les variables locales non-utilisées dans la sous-routine.

Pour traiter ce problème, plusieurs approches ont eu lieu.

Dans sa KVM [40], une machine virtuelle *Java* pour dispositifs embarqués (portée par exemple sur certains PDA), Sun supprime les sous-routines de son langage de code octet. Malheureusement, cela oblige à dupliquer du code, entraînant une croissance exponentielle du code dans le pire des cas (beaucoup moins en pratique).

La solution décrite pour la JVM par Sun dans [85, section 4.9.6] (et formalisée dans [101]) consiste à déterminer les registres utilisés par la sous-routine et à ne pas modifier les autres. Cependant, cette approche requiert de pouvoir délimiter la structure, à l'intérieur du code octet, des sous-routines et n'admet pas de solution simple sans contraintes spécifiques sur le code.

FREUND et MITCHELL [58], étendant les travaux de STATA et ABADI [115], proposent, à partir d'une sémantique opérationnelle d'un sous-ensemble de la machine virtuelle *Java*, un système de types et un algorithme de typage nécessitant, pour le traitement des sous-routines, l'étiquetage préalable du programme. Les contraintes imposées pour les sous-routines y sont restrictives.

Ainsi, aucune des ces solutions, à base de vérificateur de code octet *monovariant* (un seul état est sauvegardé pour chaque point de programme), n'apparaît satisfaisante en terme de classe de programmes acceptés.

5.1.3 Vérification de bytecode polyvariante

La variante de la vérification de code octet appelée *polyvariante* consiste à conserver un ou plusieurs états par point de programme. Elle apporte une solution efficace au problème des sous-routines décrit à la section précédente.

PC	instruction		1re passe	2e passe
0:	`jsr 20`	appel de la sous-routine au PC 20	[$\top$, $\top$]	
...				
10:	`iconst_0`	empile l'entier 0 dans la pile d'opérande	[$\top$, `RA 0`]	
11:	`istore_0`	place un entier dans le registre 0	[`int`, `RA 0`]	
12:	`jsr 20`	appel de la sous-routine au PC 20	[`int`, `RA 0`]	
13:	`iload_0`	charge un entier du registre 0	[$\top$, `RA 0`]	[`int`, `RA 12`]
14:	`ireturn`	retour de la méthode	[$\top$, `RA 0`]	[`int`, `RA 12`]
...				
20:	`astore_1`	place une adresse de retour dans le registre 1	[$\top$, `RA 0`]	[`int`, `RA 12`]
...		code n'utilisant pas le registre 0		
25:	`ret 1`	retour à l'appelant de la sous-routine	[$\top$, `RA 0`]	[`int`, `RA 12`]

TAB. 5.1 – *Exemple de code utilisant les sous-routines.*

Une même instruction peut être analysée dans plusieurs contextes différents. De manière intuitive, dans notre cas, à chaque contexte correspondrait un point d'appel possible de la sous-routine. On évite ainsi de dupliquer le code de la sous-routine au point d'appel et on garde, en l'absence de sous-routines, la même efficacité que la vérification de code octet monovariante.

Basée sur les *contours*, trace des `jsr` traversés pour aboutir à un état donné, la vérification de code octet polyvariante présente l'inconvénient majeur de ne pas terminer dans le cas de sous-routines récursives. Le javacard off-card verifier [122] de Trusted Logic, qui utilise la méthode des contours, interdit alors les sous-routines récursives, limitant ainsi les classes de programmes valides acceptés. Précisons toutefois que de telles sous-routines sont très rares et non générées par un compilateur *Java* standard.

Posegga et Vogt [18, 99] se servent de la vérification de modèle en exprimant la vérification de code octet par une formule de logique temporelle. Ils sont suivis par Coglio [41, 42] qui explorent tous les états accessibles par la machine virtuelle abstraite, exprimée par une relation de transition. En l'absence d'unification entre états lors de la vérification, le problème des sous-routines est alors évité par autant d'analyses que de chemins possibles pour accéder à une sous-routine. S'il s'agit de l'algorithme acceptant la plus large classe possible de programmes (en ne connaissant que les types des valeurs), il présente néanmoins une complexité pouvant être exponentielle en le nombre de branchements dans le programme.

Afin de réduire le nombre d'états explorés et d'obtenir une complexité satisfaisante, Leroy [84] reprend l'idée suggérée par Coglio et montre que l'on peut procéder à des étapes d'élargissement (remplacement d'un état par un état « plus grand »). Henrio et Serpette proposent enfin [65] une approche optimale en procédant à une étape d'unification entre états provenant du même contexte (même type `ReturnAddress` dans la pile d'opérande) à l'entrée de la sous-routine.

Cette dernière méthode, la plus évoluée, sera offerte dans la formalisation d'un vérificateur de code octet paramétré présentée à la section 5.3.2.

5.1.4 Vérification légère de bytecode

Les algorithmes vérifications présentés ci-dessus exigent une puissance de calcul et un espace mémoire qui, bien que raisonnables pour des ordinateurs personnels, sont hors de portée pour le microprocesseur d'une simple carte à puce. Dans le cas de *Java Card*, la vérification du programme a lieu avant son chargement sur la carte. La confiance sur le résultat de la vérification de code octet dépend donc d'un tiers que ne peut contrôler la carte à puce.

Il peut être intéressant alors de procéder à la vérification sur la carte à puce elle-même, en se servant d'algorithmes qui lui soient adaptés. Les techniques sont issues des recherches en Proof-Carrying Code de NECULA [95] et ont été spécialisées par ROSE [104] au cas de la vérification de code octet. BURDY, CASSET et REQUET ont utilisés ces techniques pour la réalisation d'un vérificateur de code octet *Java Card* embarqué [35].

La vérification légère de code octet continue à reposer sur l'intervention préalable d'un vérificateur de code octet en dehors de la carte à puce. En revanche, ce dernier produira à l'issue de la vérification un « certificat », résultat du calcul du point fixe sur les types pour chacune des instructions (l'historique obtenu dans l'algorithme de KILDALL). À partir de ce certificat et du programme, l'algorithme de vérification sur la carte à puce n'aura plus à calculer le point fixe. Il se contentera de vérifier, en une seule passe sur le programme, que le certificat fourni correspond bien à un point fixe de l'algorithme de vérification de code octet. L'économie réalisée en mémoire et en calculs est réelle, rendant possible le fonctionnement de l'algorithme sur le microprocesseur d'une carte à puce.

Certaines optimisations peuvent être effectuées sur le certificat pour en diminuer la taille. On peut par exemple ne transmettre que les types inférés pour les points de jonctions (points de sortie des instructions de branchement) du programme. En revanche, les sous-routines ne sont pas gérées correctement de par l'utilisation d'un vérificateur de code octet monovariant sur la carte à puce.

La vérification légère de code octet a été utilisée en milieu industriel pour embarquer un tel vérificateur, réalisé sous l'Atelier B, sur une carte à puce [33–35]. Elle a été également formalisée sous Isabelle/HOL et montrée correcte et complète par KLEIN et NIPKOW [77, 79].

5.1.5 Vérification de l'initialisation des objets

En dehors du typage correct des valeurs manipulées par la machine virtuelle, le vérificateur de code octet doit aussi vérifier les objets (plus précisément les instances de classe) créés durant l'exécution auront été initialisés avant d'être utilisés. Cette initialisation consiste à attribuer des valeurs particulières, déterminées par le programmeur, aux variables d'instances de l'objet. Pour une instance de classe donnée, tous les constructeurs de la relation d'héritage, depuis la classe `Object` jusqu'à la classe considérée, doivent être appelés (il s'agit de l'*appel chaîné des constructeurs*). Notons cependant que si ce mécanisme n'était pas effectué, cela ne remettrait pas en cause la sûreté du typage puisqu'à la construction tous les champs des objets créés reçoivent une valeur par défaut correspondant à leur type.

Du point de vue algorithmique, cette vérification requiert de marquer les objets comme non-initialisés à leur création et de changer leur statut ensuite. Cependant comme les références aux objets peuvent être dupliquées avant leur initialisation, il est nécessaire d'employer une forme d'analyse d'alias (aliasing analysis) pour déterminer, après l'initialisation, toutes les références dont changer le statut. Conserver avec chaque référence d'objet la position du programme où elle a été créé suffit (comme présenté à la section 3.3.1) à cette analyse.

La vérification de l'initialisation des objets est décrite de manière informelle dans [126]. FREUND et MITCHELL apportent un système de type [57] pour vérifier cette initialisation sur un sous-ensemble de la machine virtuelle *Java*. Ces travaux sont compatibles avec leur système de type pour les sous-routines [58] et leur ont permis de trouver une erreur dans une version du vérificateur de code octet distribué par Sun. BERTOT fournit une preuve sur machine [21], grâce à l'assistant de preuves COQ, de l'approche de FREUND et MITCHELL. L'équipe de Gemplus a également réalisé une preuve similaire [103] avec l'atelier B. Enfin, KLEIN a complété en Isabelle son vérificateur de code octet pour gérer l'initialisation des objets et l'appel chaîné des constructeurs [78].

Prévue dans les machines virtuelles défensives et abstraites, la vérification de l'initialisation des objets n'a pas encore été cependant prise en compte dans l'instanciation (cf chapitre 6) du vérificateur de code octet présenté ci-dessous. Nous pensons qu'en dehors du problème de l'appel chaîné des constructeurs, cette vérification s'intégrerait sans difficultés dans le cadre présenté. Il serait également envisageable de réutiliser directement en COQ les travaux de [21].

5.2 Notations de COQ

Afin de spécifier la notion de vérification de code octet dans COQ, nous avons utilisé le système de modules, récemment ajouté à COQ par CHRZĄSZCZ [37] suivant les travaux de COURANT [48].

5.2.1 Noms qualifiés

Nous avons vu dans le chapitre 4 qu'il est possible de donner à l'intérieur de fichiers (ou de sections) différents le même nom à un identificateur.

S'il y a ambiguïté dans le type de données utilisé, COQ offre la possibilité d'utiliser les *noms qualifiés*. Si un identificateur de nom `d` est défini dans les fichiers `A` et `B` alors les notations `A.d` et `B.d` permettent de différencier les

deux définitions.

Ce même mécanisme est utilisé pour l'accès aux identificateurs des modules (voir ci-dessous).

5.2.2 Les modules

Lors du développement de preuves, il est fréquent de vouloir paramétriser une théorie par des axiomes. Par exemple, dans le développement de notre vérificateur de code octet, nous aurons, entre autres, comme paramètres une notion d'état et une fonction d'exécution.

Cette fonctionnalité de paramétrisation est depuis longtemps offerte par Coq grâce au mécanisme de *section*. Dans une section, il est possible de déclarer des axiomes (hypothèses) et variables et de les utiliser alors implicitement. Une fois la section fermée, chaque théorème ou définition de la section devient paramétrée par les hypothèses ou variables dont ils dépendent. Cependant, ce mécanisme n'est pas parfait car il ne permet pas d'instancier les paramètres des sections créées. Il faut, en dehors de la section, redonner explicitement tous les paramètres à tous les éléments de la section. De plus, les théorèmes ou définitions perdent leur appartenance commune à une même structure.

La notion de module résout ces problèmes en permettant d'une part d'instancier en seule fois un ensemble de résultats et d'autre part en conservant la structure définie. Il est également possible de spécifier par l'intermédiaire d'une signature (une interface) quels sont les éléments pouvant être utilisés en dehors du module, cachant ainsi les détails d'implémentation et facilitant la réutilisation des preuves existantes.

Les modules *d'ordre supérieur*, ou *foncteurs*, rajoutent une possibilité d'abstraction en fournissant comme paramètre d'un module un autre module.

Enfin, notons que les modules permettent une gestion efficace des espaces de noms par l'utilisation des noms qualifiés (cf section 5.2.1).

En Coq, les déclarations de signature de module s'effectuent par la donnée d'une suite de paramètres (`Parameter`) et de propriétés (le mot-clé `Axiom` peut porter à confusion) entre les mots-clés `Module Type` et `End` suivis du nom donné à la signature. Une signature de module peut aussi inclure (et dans un certain sens, étendre) une autre signature de module en utilisant le mot-clé `Declare Module`.

Le mot-clé `Module` permet d'implémenter une signature de module. Il est suivi du nom attribué au module, du caractère `:` et de sa signature. Pour

chaque élément de la signature, une définition ou un lemme du même type doit être donné dans le module. La concordance entre ces types est vérifiée à la fermeture du module par **End**. Les définitions données dans le module seront masquées (on ne connaîtra à l'extérieur que leur type), sauf si la signature est précédée par `<:` au lieu de `:`.

On accède aux constructions d'un module en utilisant les noms qualifiés. Par exemple, une définition `d` d'un module `m` pourra être utilisée avec la notation `m.d`. Le mot-clé **Import** suivi d'un nom de module dispense de préfixer les constructions de ce module par son nom. Ainsi, après la commande **Import** `m`, on pourra écrire plus simplement `d` pour accéder à sa définition.

5.2.3 Quelques définitions

Nous allons introduire quelques définitions qui seront utilisées pour simplifier la présentation des hypothèses des modules de notre vérificateur de code octet.

On se donne d'abord un ensemble `A : Set`, une relation `R : A→A→Prop`, une fonction `f : A→A` et une proposition `P : A→Prop`.

On notera alors `(dec_type A)` pour signifier que l'égalité sur l'ensemble `A` est décidable, ce qui s'écrit en Coq `(a,a':A){a=a'}+{¬a=a'}`.
De manière similaire, `(dec_pred A P)` dénote la décidabilité de la proposition `P`, c'est-à-dire `(a:A){(P a)}+{¬(P a)}`.

La fonction `clos_refl_trans` des librairies Coq réalise, étant donnés un ensemble et une relation, la clôture réflexive et transitive de la relation. On peut alors exprimer que `f` est décroissante par le prédicat `monotone` qui s'écrit :
`(a:A)(clos_refl_trans ? R (f a) a)`
et que `f` est monotone (au sens large) par le prédicat `monotone` qui s'écrit :
`(a,a':A) (clos_refl_trans ? R a a') →(clos_refl_trans ? R (f a) (f a'))`.

Le prédicat `reaches` exprime quant à lui la clôture transitive de la relation `R` sous la garde `P`. Il est défini de manière inductive par :

```
Inductive reaches [A : Set; P : A→Prop; R : A→A→Prop] : A→Prop :=
  reaches_b : (a:A)(P a)→(reaches A P R a)
| reaches_s : (a,b:A)(R a b)→(reaches A P R b)→(reaches A P R a)
```

Enfin le prédicat `down_closed` exprime qu'un prédicat est fermé vers le bas (downwards-closed) :
`(a,a':A)(clos_refl_trans ? R a a')→(P a')→(P a)`.

5.3 Formalisation de vérificateurs de bytecode

Dans, cette section, nous présentons la formalisation de vérificateurs de code octet. Ceux-ci sont en effet multiples, et de plus paramétrés. Nous décrirons d'abord une abstraction BCV de la notion même de vérificateur de code octet (section 5.3.1). Nous montrerons ensuite comment à partir d'une machine virtuelle abstraite AVM et d'une structure d'historique des états `History_Struct` construire un vérificateur de code octet pour la fonction d'exécution sur la structure d'historique et pour la relation d'exécution de la machine virtuelle abstraite (section 5.3.2). Nous étendrons à nouveau ces réalisations pour fournir un vérificateur de code octet sur la fonction d'exécution d'une machine virtuelle défensive DVM, garantissant ainsi que le succès de la vérification assurera l'absence d'erreurs à l'exécution (section 5.3.3).

Toutes ces constructions sont réalisées par l'intermédiaire de modules dont l'organisation est décrite à la figure 5.1.

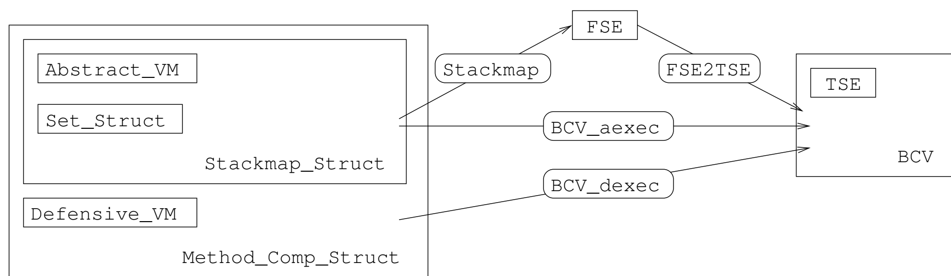

FIG. 5.1 – *Organisation des modules du vérificateur de code octet.*

Enfin, nous présenterons également la formalisation d'un vérificateur de code octet léger (section 5.3.5).

5.3.1 Abstraction de la notion de vérificateur

Nous cherchons dans cette section à satisfaire au plus près les principes de la *vérification de code octet* décrits à la section 5.1.

À partir de la composante code octet, nous dégagerons les idées d'état et de relation de transition sur ces états. Quant au mot de vérification, il suggère une notion d'erreur sur les états. Ainsi, nous allons regrouper ces trois notions dans une structure, un système de transition avec erreur.

Définition 11 *Un* système de transition avec erreur *(TSE) est donné par un type* `state` *d'*états*, une* relation d'exécution `rexec` *sur les états, et un prédicat* `err` *permettant de caractériser les états d'erreur.*
Cette structure est formalisée dans COQ *par le type de module* TSE *suivant :*

```
Module Type TSE.
Parameter state : Set.
Parameter rexec : state → state → Prop.
Parameter err   : state → Prop.
End TSE.
```

La notion de relation, capturée par la proposition `rexec` va nous permettre d'exprimer une fonction de transition qui bien que sémantiquement exécutable peut être non-déterministe et partielle.

La structure de système de transition avec erreur est la base du vérificateur de code octet, néanmoins elle ne suffit évidemment pas à le caractériser. Le vérificateur de code octet doit être capable de déterminer s'il est possible d'atteindre avec la relation de transition un état d'erreur à partir d'un état quelconque. De plus, pour être utilisable, ce test doit être décidable : le vérificateur de code octet doit donner une réponse en un temps fini.

Définition 12 *À l'intérieur d'un système de transition avec erreur, un état est dit mauvais, que l'on écrit* `bad`*, s'il est possible d'atteindre à partir de cet état et par la clôture transitive de la relation de transition un état d'erreur. Cela conduit à la définition* Coq *suivante :*

```
Definition bad := [a:state] (reaches state err_state rexec a).
```

On peut alors à partir de cette notion d'état définir très simplement la fonction d'un vérificateur de code octet :

Définition 13 *Un* vérificateur de code octet *sur un système de transition avec erreur est donné par un prédicat décidable* `check` *qui rejette tous les états mauvais.*
De manière formelle, le type de module BCV *des vérificateurs de code octet étend le module* TSE *comme suit :*

```
Module Type BCV.
Declare Module tse : TSE.
Import tse.

Parameter check : state → Prop.

Axiom check_ok  : (a:state)(check a) → ¬(bad a).
Axiom check_dec : (dec_pred ? check).
End BCV.
```

Dans cette dernière définition de module, toutes les définitions de `tse` sont accessibles à l'intérieur du type de module BCV par l'utilisation du mot-clé **`Import`**.

La manière classique de construire un vérificateur de code octet consiste à se donner un ordre bien fondé (pour lequel il n'existe pas de chaîne infinie décroissante) sur lequel l'exécution est décroissante et telle que les états d'erreurs soient fermés vers le bas. Si de plus l'exécution est déterministe, il est possible de calculer pour chaque état `a` le plus grand point fixe `b` sous `a` et il est alors suffisant de vérifier que `b` n'est pas un état d'erreur pour conclure que `a` n'est pas un état mauvais. Ces conditions sont regroupées dans la structure qui suit.

Définition 14 *Une* structure de point fixe *est donnée par un ensemble décidable d'états, une fonction d'exécution* `exec` *et un ordre bien fondé* `less` *sur ces états tels que la fonction d'exécution soit décroissante et monotone et un ensemble décidable et fermé vers le bas d'états d'erreur* `err`*.*

Formellement, cela conduit au type de module FSE *suivant :*

```
Module Type FSE.
Parameter state      : Set.
Parameter exec       : state → state.
Parameter less_state : state → state → Prop.
Parameter err_state  : state → Prop.

Axiom err_dec   : (dec_pred ? err).
Axiom err_dc    : (down_closed ? less err).

Axiom eq_dec    : (dec_type state).
Axiom less_acc  : (well_founded ? less).
Axiom exec_decr : (decreases ? less exec).
Axiom exec_mon  : (monotone ? less exec).
End FSE.
```

À titre d'exemple simple de construction de module, nous allons réaliser un module (un foncteur) FSE2TSE ayant pour type TSE et prenant comme argument un module de type FSE. Le module FSE2TSE, pour satisfaire le type de modules TSE, doit donner un terme correspondant à chacun des champs contenus dans TSE.

```
Module FSE2TSE [fse:FSE] <: TSE.
Definition state := wfs.state.
Definition err_state := wfs.err_state.
Definition rexec := [s,s':state] (exec s)=s'.
End FSE2TSE.
```

Un exemple, un peu plus compliqué cette fois, va nous permettre de construire un module FSE2BCV satisfaisant le type BCV à partir d'un module de type FSE (on le déclare alors par `Module FSE2BCV [fse:FSE] <: BCV`). Là encore, le module FSE2BCV doit donner un terme correspondant à chacun des champs contenus dans BCV. Ainsi, comme le type de module BCV annonce un

module `tse : TSE`, le module `BCV` doit aussi contenir un module de ce nom, et de ce type, ce que satisfait la déclaration `Module tse := (FSE2TSE fse)`.

La proposition `check` du module `BCV` sera quant à elle construite en calculant le plus grand point fixe `gfp` sous l'état `a` donné en argument et en vérifiant que l'on n'obtient pas un état d'erreur. Puisque l'exécution est monotone, il est clair qu'un tel test suffit à vérifier que `a` n'est pas un état mauvais (prédicat `check_ok`).

```
Definition check := [a:state]¬(err_state (gfp a)).
```

Lemme 1 *Sur une structure de point fixe, la fonction* `fixpt : state →state` *définie par :*

$$\texttt{fixpt a} = \begin{cases} \texttt{a} & \textit{si}\ \texttt{exec a = a} \\ \texttt{fixpt (exec a)} & \textit{sinon} \end{cases}$$

calcule le plus grand point fixe sous l'état `a`.

Ce calcul de point fixe se réalise, sans grande difficulté, par récursion bien fondée sur l'ordre sur les états (on notera qu'il s'agit de la seule fois dans le développement où apparaît ce type de récursion). Enfin, avec la calculabilité de `fixpt` et la décidabilité de `err_state`, on satisfait le prédicat `check_dec` du module `BCV` et donc la signature donnée au module `FSE2BCV`.

5.3.2 Instanciation paramétrique à la machine abstraite

Dans cette section, nous spécifions notre définition de vérificateur de code octet en la rapprochant de la définition d'une machine virtuelle. Le vérificateur de code octet rejettera alors les programmes qui se comportent mal lorsqu'ils sont exécutés sur une machine virtuelle abstraite. Il sera paramétré par une fonction d'optimisation nous permettant à partir du même cadre formel d'obtenir un vérificateur de code octet monovariant ou polyvariant (voir sections 5.1.1 et 5.1.3).

Abstraction d'une machine virtuelle abstraite

Nous commençons par la définition d'une machine virtuelle abstraite, sur laquelle s'exécutera le vérificateur de code octet.

Définition 15 *Une machine virtuelle abstraite (AVM) est donnée par un type décidable et ordonné d'états* `state` *comportant un ensemble décidable*

d'états d'erreur `err` *fermé vers le bas, par un type décidable* `loc` *d'emplacements de programme, par une fonction d'exécution* `exec`*, par une fonction successeur* `succs` *calculant la prochaine instruction à exécuter et enfin par une énumération* `locs` *des emplacements du programme.*

Formellement, cela conduit à la définition suivante du module `AVM` *:*

```
Module Type AVM.
Parameter state      : Set.
Axiom eq_state_dec   : (dec_type state).

Parameter less_state : state → state → Prop.

Parameter err_state  : state → Prop.
Axiom err_state_dec  : (dec_pred ? err_state).
Axiom err_state_dc   : (down_closed ? less_state err_state).

Parameter loc        : Set.
Axiom eq_loc_dec     : (dec_type loc).

Parameter exec       : loc → state → state.

Parameter succs      : loc → state → (list loc).
Parameter locs       : (list loc).
End AVM.
```

Le vérificateur de code octet se servira alors des états de cette machine virtuelle abstraite dans sa structure d'historique associant à chaque point de programme un état dans le cas d'une analyse monovariante ou un ensemble d'états dans le cas d'une analyse polyvariante.

Structure d'historique

Pour suivre alors les suggestions de [65], et rendre notre analyse plus modulaire, nous abstrayons la structure d'historique utilisée par le vérificateur de code octet par une structure comparable à un ensemble. Cette structure abstraite sera utilisée dans le suite pour les calculs du BCV et il suffira au moment de l'utilisation de ce dernier d'instancier cette structure suivant le type de vérification voulue. Nous fournissons deux instances de cette structure correspondant aux analyses monovariante et polyvariante.

Définition 16 *On définit une structure d'historique abstraite à partir d'un type d'ensemble décidable* `A`*, par un constructeur* `hist` *de notre type d'ensemble que l'on veut aussi décidable, par une fonction d'injection* `hist_c` *de* `A` *dans* `(hist A)`*, par un constructeur d'ordre* `hist_less`*, par un prédicat d'appartenance* `hist_In` *et par la déclaration d'une fonction itérative décroissante* `hist_fold_right`*. On définit de plus un prédicat d'existence*

hist_ExistsS *à partir de* hist_In *et on demande que ce prédicat soit décidable et fermé vers le bas.*

Formellement, cela conduit à la définition suivante du module History_Struct *:*

```
Module Type History_Struct.

Parameter A : Set.
Axiom A_dec : (dec_type A).

Parameter hist : Set → Set.
Axiom hist_dec : (dec_type (hist A)).

Parameter hist_c : A→(hist A).
Parameter hist_less : (le:A→A→Prop)(hist A)→(hist A)→Prop.

Parameter hist_In : A→(hist A)→Prop.
Axiom hist_In_hist_c : (x:A)(hist_In x (hist_c x)).

Parameter hist_fold_right : (B:Set)(A→B→B)→B→(hist A)→B.
Axiom hist_fold_right_decr : (B:Set; f:(A→B→B); less_B:(B→B→Prop))
  ((a:A)(decreases ? less_B (f a)))→
  ((a:(hist A))(decreases ? less_B ([b:B] (hist_fold_right ? f b a)))).

Syntactic Definition hist_ExistsS := [P:(A→Prop)][s:(hist A)]
  (EX a | (P a) ∧ (hist_In a s)).
Axiom hist_ExistsS_dec : (P:A→Prop)
  (dec_pred ? P)→(dec_pred ? (hist_ExistsS P)).
Axiom hist_ExistsS_dc : (le:A→A→Prop)(P:A→Prop)
  (down_closed ? le P)→(down_closed ? (hist_less le) (hist_ExistsS P)).

End History_Struct.
```

La structure d'historique pour les états est alors directement construite sur la structure d'historique abstraite en prenant comme ensemble décidable les états de la machine virtuelle abstraite. À ce point, il faut aussi rajouter la fonction d'unification entre un état de la machine abstraite et un élément de la structure d'historique.

Définition 17 *Une structure d'historique d'états est donné par une machine virtuelle abstraite* avm*, par une structure d'ensemble* hist_Struct*, par un élément maximal* top_st *de l'ordre sur les états, par une preuve* less_state_acc *de bonne fondation de ce dernier ordre et par une fonction d'unification décroissante* unify *sur la structure d'ensemble.*

Formellement, cela conduit à la définition suivante du module Stackmap_Struct *:*

```
Module Type Stackmap_Struct.
Declare Module avm : AVM.
```

```
Import avm.

Declare Module hist_Struct : History_Struct
  with Definition A := avm.state
  with Definition A_dec := avm.eq_state_dec.
Import hist_Struct.

Parameter top_st : state.
Axiom le_top_st : (a:state)(clos_refl_trans ? less_state a top_st).

Axiom less_state_acc : (well_founded ? less_state).

Axiom unify : state → (hist state) → (hist state).
Axiom unify_decr : (s:state)(s':(hist state))
  (clos_refl_trans ? (hist_less less_state) (unify s s') s').
Axiom unify_eq : (s:state; s':(hist state))
(unify s s')=s'→
(EX y | (clos_refl_trans ? avm.less_state y s) ∧ (hist_In y s')).

Axiom exec_mon : (l:loc)(monotone ? less_state (exec l)).
End Stackmap_Struct.
```

Le module `Stackmap_Struct` contient maintenant tout ce qui est nécessaire à l'obtention d'un BCV. Le foncteur de module `Stackmap` construit alors à partir d'un module de type `Stackmap_Struct` un module de type `FSE`, c'est-à-dire une structure de point fixe avec erreur. Ce dernier pourra alors être fourni au foncteur de module `FSE2BCV` pour obtenir un module de type `BCV`.

Construction du BCV pour la machine abstraite

Les états que manipule le module `Stackmap` représentent un historique des états de la machine virtuelle pour un certain nombre de points de programme (intuitivement, tous les points de programme de la méthode à analyser). Ils sont alors naturellement définis, à partir d'un module de type `Stackmap_Struct`, par :

```
Definition stackmap := (list (avm.loc * (hist avm.state))).
Definition state := stackmap.
```

Fonction d'exécution sur la structure d'historique La fonction d'exécution sur ces états nécessite un certain nombre d'itérations. En suivant l'algorithme de Kildall, il s'agit d'exécuter chacun des états de la structure d'historique à chacun des points de programme et d'unifier le résultat de l'exécution avec les états successeurs. Ainsi pour définir la fonction `exec` du

type de module FSE nous effectuons :

- une itération sur tous les points de programme (réalisé par un map) ;
- à un point de programme donné, une itération sur tous les états de la structure d'historique (réalisé par un hist_fold_right) ;
- à un point de programme et à un état de la structure d'historique à ce point, un pas d'exécution de la machine abstraite sur cet état et une unification du résultat avec toutes les structures d'historique situées à un successeur de l'état (réalisé par un fold_right).

Cette fonction d'exécution peut être optimisée en marquant par un booléen les états nécessitant une nouvelle passe d'exécution (ceux pour lesquelles la fonction d'unification a rendu à la passe précédente un résultat nouveau). Il s'agit d'une optimisation relativement simple, mais qui n'est pas présentée ici.

On montre que cette fonction d'exécution est décroissante, sous l'hypothèse fournie par le module Stackmap_Struct que la fonction d'exécution de la machine virtuelle abstrait est elle-même décroissante. Pour compléter et satisfaire le type de module FSE, on définit les états d'erreur comme les états où il existe à l'intérieur d'une structure d'historique un état d'erreur de la machine virtuelle abstraite. Enfin, on montre les propriétés posées sur ce prédicat err_state.

S'il l'on applique directement le foncteur FSE2BCV au module construit par le module Stackmap, on obtient un BCV sur des états représentant des structures d'historique et se rapportant à la fonction d'exécution sur ces états. Or on souhaite disposer d'un BCV travaillant directement sur un état de la machine virtuelle abstraite et sur sa fonction d'exécution. Il suffirait ainsi de vérifier l'état initial de la méthode que l'on souhaite analyser pour décider s'il peut conduire à un état d'erreur ou non.

Relation de transition sur la machine abstraite Ce vérificateur de code octet sur les états de la machine virtuelle abstraite est construit par le foncteur de module BCV_aexec, de type BCV, à partir d'un module de type Stackmap_Struct.

Pour obtenir un type de module BCV, il faut fournir un module de type TSE. Dans notre cas, ce système de transition sera le suivant :

```
Module tse.
Definition state := avm.loc*avm.state.
Definition rexec := [x,y:state]
  ((avm.exec (Fst x) (Snd x)) = (Snd y)) /\
    (In (Fst y) (avm.succs (Fst x) (Snd x))).
Definition err_state := [x:state](avm.err_state (Snd x)).
End tse.
```

On travaille alors sur des états constitués d'une indication d'emplacement de programme et d'un état de la machine virtuelle abstraite. La relation de transition est bien celle donnée par la fonction d'exécution de la machine virtuelle abstraite pour deux états successeurs l'un de l'autre.

Pour faire le lien avec notre BCV sur les états avec structure d'historique (`stackmap`), il nous faut utiliser un de ces états. La fonction `mk_sm` construit alors à partir d'un état de la machine virtuelle un état de type `stackmap` en le plaçant à sa bonne position et en plaçant aux autres positions l'élément maximal du treillis :

```
Definition mk_sm := [s:loc*avm.state]
(map ([m:loc] (m, (Cases (eq_loc_dec m (Fst s)) of
                      (left  _) ⇒ (hist_c (Snd s))
                    | (right _) ⇒ (hist_c top_st)
                    end))) locs).
```

Si `sms` est notre module de type `Stackmap_Struct` alors on construit la structure de point fixe qui lui correspond par le module `Stackmap` et le BCV associé à cette structure de point fixe :

```
Module fps := (Stackmap sms).
Module bcv := (WF_BCV fps).
```

La fonction `check` satisfaisant la signature du module BCV pour la relation d'exécution de la machine virtuelle abstraite se définit par rapport à l'exécution sur la structure d'historique :

```
Definition check :=
[s:state] ¬(fps.err_state (bcv.fixpt (mk_sm s))).
```

On arrive finalement à montrer le lemme principal de ce module assurant que pour tout état de la machine virtuelle abstraite la vérification échoue s'il est possible d'atteindre à partir de cet état un état d'erreur :

```
Lemma check_ok : (a:state)(check a) → ¬(bad a).
```

Le module `BCV_aexec` correspond à une vérification de code octet telle qu'elle est normalement attendue. Cette vérification est paramétrée par le type d'analyse souhaitée. De plus, toutes les preuves sur la vérification de code octet sont apportées et font partie inhérente du module.

Dans la section suivante, nous allons voir comment étendre encore ce vérificateur de code octet pour aboutir à une vérification de la machine virtuelle défensive.

5.3.3 Instanciation à la machine défensive

Nous souhaitons maintenant construire un vérificateur de code octet pour la machine défensive. Ce dernier devra, étant donné un état de la machine virtuelle défensive, indiquer si l'exécution conduit à un état d'erreur. L'idée est de se servir du vérificateur de code octet sur la machine virtuelle abstraite et de montrer que les programmes qui passent cette vérification n'engendrent pas d'erreur à l'exécution de la machine virtuelle défensive.

Abstraction d'une machine virtuelle défensive

Nous sommes d'abord amenés à définir une machine virtuelle défensive :

Définition 18 *Une machine virtuelle défensive (DVM) est donnée par un type d'états* `state`*, par un type de contexte d'exécution* `frame`*, par un accesseur à la pile des contextes d'exécution* `getstack` *et par une fonction d'exécution* `exec`*. Un prédicat* `err_frame` *permet de déterminer les contextes d'exécution incorrects.*

Formellement, cela conduit à la définition suivante du module `DVM` *:*

```
Module Type DVM.

Parameter state : Set.
Parameter frame : Set.
Parameter getstack : state→(list frame).
Parameter exec : state→state.

Parameter err_frame : frame→Prop.

End DVM.
```

On construit un type de module `Method_Comp_Struct` contenant les fonctions, prédicats et obligations de preuve qu'il est nécessaire de fournir pour aboutir au vérificateur de code octet sur la machine virtuelle défensive.

Tout d'abord, ce type de module contient des déclarations de module `dvm` de type `DVM` et `sms` de type `Stackmap_Struct` pour construire le vérificateur de code octet abstrait et disposer de la machine virtuelle `avm`.

On se donne de plus une fonction `getcf` qui classifie les états de la machine défensive selon le résultat d'un pas d'exécution. On distingue alors quatre cas d'exécution : intra-procédurale `sameframe` (*i.e.* n'agit que sur le contexte d'exécution courant comme une instruction arithmétique, de branchement, etc) ; appel de méthode `invoke` ; retour de méthode `return` ; lancement d'exception `exception`. Cette fonction nous sera utile quand il s'agira

de distinguer le type d'action en cours de la machine virtuelle pour justifier la technique de vérification méthode par méthode (compositionnelle).

Correspondance entre les machines virtuelles

Il est nécessaire pour raisonner simultanément sur les machines virtuelles défensives et abstraites d'établir un certain nombre de correspondances entre ces deux machines.

Fonction d'abstraction La principale est fournie par la fonction d'abstraction `alpha` de `dvm.frame` vers `avm.loc*avm.state`. Cette dernière est étendue aux piles de contextes d'exécution par la fonction `beta` de `dvm.state` vers `avm.loc*avm.state` en abstrayant par `alpha` le haut de la pile (une valeur par défaut est retournée si la pile est vide). On établit également la correspondance entre les états d'erreur des deux machines virtuelles : si un état est un état d'erreur pour la machine virtuelle défensive, alors son abstraction est aussi un état d'erreur pour la machine virtuelle abstraite.

Correspondance des exécutions On cherche ensuite à établir une relation entre les exécutions sur les machines défensive et abstraite. Dans le cas d'une instruction intra-procédurale, cette correspondance (aussi appelé validation croisée des machines virtuelles défensive et abstraite) sera établie principalement grâce au diagramme de commutation de la figure 5.2.

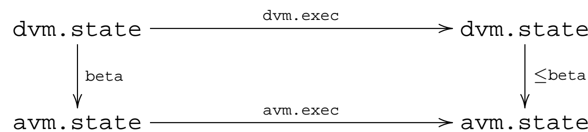

FIG. 5.2 – *Diagramme de commutation des exécutions défensive et abstraite.*

Ce diagramme s'exprime en COQ de la manière suivante :

```
Axiom diagram_sf: (a:dvm.state)
  (safe_memory a)→
  (getcf a) = sameframe→
  (le_avm_state (aexec (beta a)) (Snd (beta (dvm.exec a)))).
```

où `safe_memory` est un prédicat sur les états décrit plus bas et `le_avm_state` la fermeture réflexive et transitive de l'ordre sur les états abstraits.

Dans le cas d'un appel de méthode, il est nécessaire de décomposer la fonction d'exécution abstraite. En effet, celle-ci se fait en un seul pas dans la version abstraite alors que dans la version défensive elle conduit à la modification du contexte courant, à la création et à l'exécution d'un nouveau contexte et enfin au retour de ce contexte. En introduisant les fonctions `aexec_add` et `aexec_ret`, on sera alors en mesure de faire correspondre à une contexte d'exécution en cours d'appel de méthode un état abstrait. Ces fonctions ne sont appelées que si l'instruction courante est un appel de méthode (`getcf` rend `invoke`) et on requiert dans ce cas là que la composition `aexec_ret` et `aexec_add` soit égale à la fonction d'exécution abstraite `avm.exec`. On se donne également une fonction `init` qui rend étant donné une emplacement de programme, l'état initial (l'état correspondant à la première instruction à exécuter) associé à la méthode ou au gestionnaire d'exception visé.

Pour notre vérificateur de code octet sur la machine virtuelle défensive, il faut vérifier maintenant que, dans le cas d'un appel de méthode, l'abstraction de l'état résultat de l'appel de méthode dans le nouveau contexte est plus grand que l'état initial pour la méthode appelée. Cela est normalement assuré par la vérification des arguments vis-à-vis de la signature de la méthode lors de son appel.
La propriété correspondante s'écrit en Coq :

```
Axiom diagram_ivk : (a:dvm.state)(f:dvm.frame)(lf:(list dvm.frame))
  (safe_memory a)→
  (getcf a) = invoke→
  (dvm.getstack a)=(cons f lf)→
  (EX f' |
    ((dvm.getstack (dvm.exec a))=(cons f' (cons (aexec_add f) lf)) ∧
      (le_avm_state (Snd (init (Fst (beta (dvm.exec a)))))
                              (Snd (beta (dvm.exec a)))))).
```

où l'on relie également les piles de contexte par l'utilisation de `aexec_add`.

De manière similaire pour le cas d'un retour de méthode, il nous faut assurer que le contexte ayant appelé la méthode est, après exécution du retour, plus grand que l'exécution de `aexec_ret` sur l'abstraction de ce même contexte avant exécution :

```
Axiom diagram_ret : (a:dvm.state)(f:dvm.frame)(lf:(list dvm.frame))
  (getcf a) = return →
  (dvm.getstack (dvm.exec a)) = (cons f lf) →
  (EX f' | (EX f'' | (dvm.getstack a)=(cons f' (cons f'' lf)) ∧
    (le_astate (aexec_ret (alpha f'') (alpha f')) (alpha f))
```

Enfin, dans le cas d'une exception, il faut vérifier que l'abstraction de l'état résultant du rattrapage de l'exécution (situé au début d'un gestionnaire d'exceptions) est plus grand que l'état initial abstrait correspondant à ce gestionnaire d'exceptions.

```
Axiom diagram_exc : (a:dvm.state)(f:dvm.frame)(lf,lf':(list dvm.frame))
  (safe_memory a)→
  (getcf a) = exception→
  (dvm.getstack a)=(cons f (app lf lf'))→
  (EX f' |
    ((dvm.getstack (dvm.exec a))=(cons f' lf')) ∧
      (le_avm_state (Snd (init (Fst (beta (dvm.exec a)))))
                             (Snd (beta (dvm.exec a)))))).
```

À ces obligations de preuves, relativement complexes mais indispensables, viennent s'ajouter d'autres propriétés triviales sur la bonne formation de la pile de contexte d'exécution de méthode selon le type d'instruction exécutée et des propriétés assurant un flot de programme similaire entre les deux machines virtuelles.

Prouver ces propriétés, en particulier celles sur la machine virtuelle défensive, requiert d'assurer en pré-condition de ces propriétés une notion de zone mémoire bien formée. Cette notion sera capturée par la déclaration de prédicat `safe_memory`. De manière intuitive, ce prédicat assurera que les objets trouvés dans le tas de la machine virtuelle correspondent bien au type des valeurs qui leur font référence. Ce prédicat doit être un invariant de l'exécution et il faut donc apporter la preuve suivante :

```
Parameter well_formed_memory_mon : (s:dvm.state)
  (well_formed_memory s)→(well_formed_memory (dvm.exec s)).
```

Au total, le type de module `Method_Comp_Struct` contient environ une dizaine de preuves à apporter. Ces preuves serviront à assurer un invariant essentiel de l'exécution défensive en rapport avec le vérificateur de code octet abstrait et présenté ci-dessous.

Notion d'état sûr

Nous passons maintenant à la définition de la dernière notion essentielle dans la construction de notre vérificateur de code octet pour la machine défensive, la notion de *sûreté* d'un état.

Définition 19 *Soit un état a de la machine virtuelle abstraite. Soit res_bcv le résultat du calcul du vérificateur de code octet abstrait à partir de l'état initial de la méthode correspondant à a. Alors a est dit sûr (safe), si dans la structure d'historique de res_bcv à la position de a il existe un état plus petit que a.*

```
Definition safe_astate := [s:avm.locs*avm.state]
(EX ms | ((sm_at_loc (res_bcv (Fst s)) (Fst s))=(Some ? ms) ∧
  (hist_ExistsS ([a:state] (le_avm_state a (Snd s))) ms)))
∧ (In (Fst s) avm.locs).
```

où `sm_at_loc` *est une fonction partielle retournant la structure d'historique du vérificateur de code octet à une position donnée et où* `res_bcv` *calcule, étant fixé un module* `sms` *de type* `Stackmap_Struct`*, le résultat de l'exécution du vérificateur de code octet abstrait à partir de l'état initial correspondant au point de programme donné en paramètre :*

```
Module bcva := (BCV_aexec sms).
Import bcva.

Definition res_bcv := [l:loc] (bcv.fixpt (mk_sm (init l))).
```

Ainsi, le vérificateur de code octet calculant le plus petit point fixe des types utilisés dans le programme, si le vérificateur de code octet ne contient pas d'état d'erreur alors l'état sûr n'est pas lui-même un état d'erreur.

Cette notion est étendue aux contextes d'exécution de la machine virtuelle défensive. Pour le contexte du haut de la pile, on définit simplement :

```
Definition safe_top_dframe := [d:dvm.frame]
 (safe_astate (alpha d)).
```

Pour les autres contextes de la pile, ils doivent résulter d'un appel de méthode et être sûrs avant ce dernier. On écrit ainsi :

```
Definition safe_ivk_dframe := [d:dvm.frame]
 (EX d' | (alpha d)=(aexec_add (alpha d'))∧(safe_top_dframe d')).
```

Une pile de contexte est donc sûre (`safe_stack`) si le contexte du haut vérifie `safe_top_dframe` et les suivants `safe_ivk_dframe`.
Finalement un état de la machine virtuelle défensive est sûr si sa pile de contextes d'exécution est sûre et la mémoire est bien formée.

```
Definition safe_dstate := [s:dvm.state]
 (safe_stack (dvm.getstack s))∧(well_formed_memory s).
```

Lemme 2 *À partir des propriétés décrites pour le module* `Method_Comp_Struct` *et par analyse de cas suivant le type des instructions, on montre que le prédicat* `safe_dstate` *est un invariant de l'exécution de la machine virtuelle défensive.*

```
Lemma safe_exec : (s:dvm.state)
  (safe_dstate s)→(safe_dstate (dvm.exec s)).
```

Construction du vérificateur de bytecode

On construit un module `BCV_dexec` de type `BCV` pour la relation d'exécution de la machine virtuelle défensive. La fonction `check` de ce module se définit maintenant simplement par rapport à la notion de sûreté :

```
Definition check := safe_dstate.
```

Sous l'hypothèse que pour toutes les méthodes du programmes, le résultat de la vérification de code octet abstraite ne contient pas d'état d'erreur et grâce au Lemme 2, on satisfait la propriété `check_ok` du module `BCV` :

```
Lemma check_ok : (a:state)(check a) → ¬(bad a).
```

Cette dernière propriété nous assure que pour tout état de la machine virtuelle défensive la vérification échoue s'il est possible d'atteindre à partir de cet état un état d'erreur. Le vérificateur de code octet défensif est construit!

Lien avec la machine virtuelle offensive

Il est possible d'exploiter directement le résultat obtenu sur le vérificateur de code octet défensif pour exprimer une des propriétés principales du vérificateur de code octet vis-à-vis des machines virtuelles défensive et offensive : l'égalité de l'exécution de celles-ci si la vérification est passée.

Il est nécessaire pour exprimer cette propriété de donner la notion de machine virtuelle offensive.

Définition 20 *Une machine virtuelle offensive (OVM) est donnée par un type d'états* `state` *et par une fonction d'exécution* `exec`*.*

Formellement, cela conduit à la définition suivante du module `OVM` *:*

```
Module Type OVM.

Parameter state : Set.
Parameter exec : state→state.

End OVM.
```

On étend alors le type de module `Method_Comp_Struct` en le type de module `Off_Method_Comp_Struct` en y rajoutant une machine virtuelle offensive `ovm` et une fonction d'abstraction `alpha_off : dvm.state→ovm.state`. Enfin, on exprime la validation croisée des machines défensives qui se traduit par le diagramme de la figure 5.3 et par la propriété Coq :

```
Axiom diagram_off : (s:dvm.state)
¬(ExistsS ? dvm.err_frame (dvm.getstack s))→
(alpha (dvm.exec s))=(ovm.exec (alpha s))
```

où la prémisse précise que `s` n'est pas un état d'erreur.

Lemme 3 *À partir des propriétés décrites pour le module* `Off_Method_Comp_Struct` *et du vérificateur de code octet construit avec le*

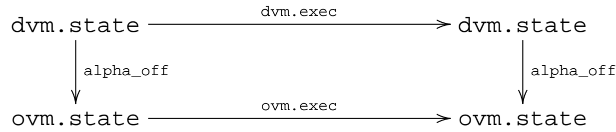

FIG. 5.3 – *Diagramme de commutation des exécutions défensive et offensive.*

module `BCV_dexec`*, on montre que pour tout état* `s`*, si la vérification de* `s` *réussit alors les résultats des exécutions offensive et défensive sont égaux.*

Formellement, cette propriété s'exprime en COQ *par :*

```
Lemma safe_oexec : (s:dvm.state)
(check s)→
(mws.alpha (dvm.exec s))=(ovm.exec (alpha_off s)).
```

La preuve s'effectue très simplement par application du Lemme 2 et de la propriété de validation croisée de la figure 5.3.

5.3.4 Différents paramétrages

Nous fournissons ici différents paramétrages du vérificateur de code octet correspondant aux analyses monovariante, polyvariante et hybride. Ceux-ci simplifient la tâche de fournir le module de type `Stackmap_Struct` utilisé pour le vérificateur de code octet sur la fonction d'exécution de la machine virtuelle abstraite en ne requérant que les contraintes nécessaires sur cette machine virtuelle. Ces paramétrages se veulent donc comme des points d'entrée directs dans la formalisation et dispensent l'utilisateur de la connaissance de son organisation.

Vérification monovariante

Dans le cas d'une analyse monovariante, l'ensemble d'états manipulé par la structure d'historique est réduit aux états eux-mêmes. À ce type d'analyse correspond donc une instance triviale `Id_History_Struct` du module `History_Struct` où l'ensemble abstrait est l'ensemble des états lui-même. L'extension de l'ordre sur l'ensemble abstrait `hist_less` est alors directement l'ordre sur les états. La relation d'appartenance `hist_in` est l'égalité. Enfin, l'application `hist_fold_right` d'une fonction f est l'application de f elle-même.

La structure d'historique fixée de cette manière, les données nécessaires sur la machine virtuelle abstraite pour construire un module de type `Stackmap_Struct` sont rassemblées dans la structure suivante :

Définition 21 *Une* structure de vérification monovariante *est définie par la donnée d'une machine virtuelle abstraite, d'une fonction d'unification décroissante sur les états, d'un ordre bien fondé sur ces derniers avec un élément maximum et d'une fonction d'exécution monotone.*

De manière formelle :

```
Module Type MVS.
Declare Module avm : AVM.
Import avm.
Parameter unify  : state → state → state.
Axiom unify_decr : (s:state)(decreases ? less_state (unify s)).
Axiom unify_eq   : (s:state; s':state) (unify s s')=s' →
                   (clos_refl_trans ? less_state s' s).

Axiom less_state_acc : (well_founded ? less_state).

Parameter top_st : state.
Axiom le_top_st  : (a:state)(clos_refl_trans ? less_state a top_st).

Axiom exec_mon : (l:loc)(monotone ? less_state (exec l)).
End MVS.
```

Pour faire le lien avec la structure `Stackmap_Struct`, nous fournissons un foncteur de module (`MVS2SMS`). Ce foncteur utilise la structure `Id_History_Struct` et les données d'un module de type `MVS` passé en argument pour construire un module de type `Stackmap_Struct`.

Vérification polyvariante

Dans le cas d'une analyse polyvariante, l'unification entre un état provenant d'un pas d'exécution et la structure d'historique abstrait consiste à rajouter cet état à l'ensemble. Pour que la vérification termine, il faut donc montrer que l'ensemble d'historique abstrait est fini. Une des possibilités pour cela serait de montrer que l'ensemble des types manipulés par la machine virtuelle est fini. Ainsi, puisque le nombre de variables locales et la taille maximale de la pile d'opérandes sont connus, l'ensemble des états manipulés par la machine virtuelle est lui aussi fini. Cette solution est cependant difficile à mettre en œuvre car elle requiert soit une définition du système de types de la machine virtuelle induisant par construction un ensemble fini de types (ce n'est pas le cas pour notre système de type) soit une énumération des types manipulés par la machine virtuelle et la preuve que cette énumération est invariante par exécution. Pour assurer que l'ensemble

d'historique abstrait est fini, nous préférons fixer une taille maximale pour ce dernier et y placer un état d'erreur si cette taille est atteinte (afin d'assurer la correction de l'analyse). Cette solution, plus simple pour les preuves, présente également l'avantage de pouvoir borner l'espace mémoire utilisé pour la vérification d'un programme.

Définition 22 *Une* structure de vérification polyvariante *est définie par la donnée d'une machine virtuelle abstraite, d'une taille maximale pour les ensembles manipulés, d'un état d'erreur, d'un état maximum et d'une fonction d'exécution monotone.*

De manière formelle :

```
Module Type PVS.
Declare Module avm.
Import avm.

Parameter max_length_set : nat.

Parameter err : state.
Axiom err_state_err : (err_state err).

Parameter top_st : state.
Axiom le_top_st  : (a:state)(clos_refl_trans ? less_state a top_st).

Axiom exec_mon : (l:loc)(monotone ? less_state (exec l)).
End PVS.
```

Pour ce type de vérification, nous instancions l'ensemble d'états de la structure d'historique par une structure d'ensemble `List_set_History_Struct` utilisant une représentation des ensembles mathématiques sous forme de listes (librairie Coq `ListSet`). L'extension de l'ordre sur l'ensemble abstrait `hist_less` est l'ordre sur les cardinaux des ensembles. La relation d'appartenance `hist_in` est l'appartenance à un ensemble. Enfin, à l'application `hist_fold_right` est associée l'application `fold_right` sur les listes.

Comme pour l'analyse monovariante, nous fournissons un foncteur de module `PVS2SMS` prenant en argument un module de type `PVS` et satisfaisant le type de module `Stackmap_Struct`.

Vérification hybride

La vérification hybride est une variante des vérifications monovariante et polyvariante où une l'unification entre états n'intervient que dans des cas spécifiques discriminés par une fonction d'optimisation (on cherchera par exemple avant d'unifier si les états ont des adresses de retour identiques).

Une structure de vérification hybride contiendra donc, outre la fonction d'optimisation, l'union des éléments des vérifications monovariante et polyvariante.

Définition 23 *Une* structure de vérification hybride *est donnée par une machine virtuelle abstraite, une fonction booléenne entre états, une fonction d'unification décroissante sur les états, un ordre bien fondé sur ces derniers avec un élément maximum, une fonction d'exécution monotone, une taille maximale pour les ensembles manipulés, un état d'erreur, un état maximum et enfin une fonction d'exécution monotone.*

De manière formelle,

```
Module Type HVS.
Declare Module avm.
Import avm.
Parameter unify  : state → state → state.
Axiom unify_decr : (s:state)(decreases ? less_state (unify s)).
Axiom unify_eq   : (s:state; s':state) (unify s s')=s' →
                   (clos_refl_trans ? less_state s' s).

Parameter opt_unify : state → state → bool.

Axiom less_state_acc : (well_founded ? less_state).

Parameter top_st : state.
Axiom le_top_st : (a:state)(clos_refl_trans ? less_state a top_st).

Axiom exec_mon : (l:loc)(monotone ? less_state (exec l)).

Parameter max_length_set : nat.

Parameter err : state.
Axiom err_state_err : (err_state err).
End HVS.
```

Pour construire un module de type `Stackmap_Struct` à partir d'un module de type `HVS`, le foncteur de module `HVS2SMS` utilise (comme pour l'analyse polyvariante) la structure d'ensemble `List_set_History_Struct`. La fonction d'unification du module `Stackmap_Struct` est définie en unifiant ou en rajoutant à la structure d'historique les états provenant de l'exécution selon la fonction `opt_unify`. La taille maximale des ensemble `max_length_set` et les propriétés sur la fonction d'unification garantissent la terminaison de l'algorithme.

5.3.5 Un vérificateur de bytecode léger

Nous présentons ici un modèle abstrait de vérificateur de code octet léger (voir section 5.1.4), à la manière du modèle présenté à la section 5.3.1.

Ce modèle se base sur une notion de certificat, qui sera la preuve transmise avec le programme, de la sûreté du typage.

Définition 24 *Un* vérificateur de code octet *sur un système de transition avec erreur est donné par un ensemble* `cert` *de certificats, un prédicat décidable* `check_light` *sur les états et les certificats qui rejette tous les états mauvais.*
De manière formelle, le type de module `LBCV` *des vérificateurs de code octet légers étend le module* `TSE` *comme suit :*

```
Module Type LBCV.
Declare Module tse: TSE.
Import tse.
Parameter        cert : Set.
Parameter check_light : cert → state → Prop.

Axiom check_light_ok  : (a:state)(c:cert)(check_light c a) → ¬(bad a).
Axiom check_light_dec : (c:cert)(dec_pred ? (check_light c)).
End LBCV.
```

Il est possible de construire un vérificateur de code octet léger à partir d'une structure de point fixe en prenant les états eux-mêmes comme certificat. La vérification légère pour un état `a` et certificat `c` est alors réalisée en vérifiant que `c` :

- est plus petit que `a` ;
- est un point fixe pour `exec` ;
- n'est pas un état d'erreur.

Ces opérations sont effectuées par le module `WF_LBCV` prenant en argument un module de type `FSE` et satisfaisant le type `LBCV`.

La vérification de code octet légère est correcte et complète vis-à-vis de la vérification de code octet.

Lemme 4 *Sur une structure de point fixe avec erreurs, pour tout état* `a` *et* `c`*, les propriétés suivantes sont vérifiées :*

Correction : `(check_light a c) →(check a).`
Complétude : `(check a) →(check_light a (fixpt a)).`

Notons finalement que cette notion de vérification de code octet légère peut être raffinée, plus particulièrement en qui concerne la notion de certificat. Il est possible et il apparaît facile de ne transmettre dans le certificat, comme réalisé dans [35, 78], que l'indication des types calculés au niveau des points de jonction du programme. Les types manquant peuvent alors être calculés en une seule passe sur le programme pendant la vérification du

certificat. Cette approche est également prouvée correcte et complète par Klein et Nipkow [77, 79].

5.4 Conclusions

Nous avons conçu un cadre générique dans lequel nous avons établi la correction d'un vérificateur paramétrisé de code octet et justifié la technique compositionnelle de la vérification (traitement méthode par méthode et gestion des exceptions). Ce cadre fournit une interface commode par laquelle construire un vérificateur certifié de code octet pour les plates formes *Java* ou *Java Card*.

Un des aspects pratique et intéressant de notre travail est de se reposer sur le système de modules de la version 7.4 de Coq. Notre expérience montre que les modules peuvent avantageusement être exploités pour fournir des développements structurés. Cependant, nous avons trouvé un certain nombre de situations où les modules de première classe auraient été utiles (les travaux de Augustsson [3] et de Coquand, Pollack et Takeyama [47] montrent qu'il est possible de considérer les modules comme des types dépendants d'enregistrement). De même, les modules bien qu'apportant des fonctionnalités similaires aux objets des langages à objets n'offrent pas la même facilité d'utilisation.

Travaux à venir Il est possible d'améliorer encore le travail présenté selon trois directions :

Vérificateurs de code octet optimisés. Afin d'améliorer l'efficacité du vérificateur de code octet extrait de notre modèle, il semble important de formaliser et de prouver correct un certain nombre d'optimisations, par exemple celles proposées dans [65];

Vérificateurs de code octet légers optimisés. Afin de minimiser l'utilisation de l'espace mémoire, les implémentations de vérificateurs de code octet légers reposent uniquement sur les annotations aux points de jonction du programme. À partir de ceux-ci, la structure d'historique (qui sert de certificat dans nos preuves) peut être reconstruite. Il semble intéressant d'étendre notre travail dans cette direction. Nous nous attendons à ce que notre formalisme algébrique rendre cette tâche relativement aisée.

Vérificateurs de code octet renforcés. Comme souligné dans l'introduction, notre travail fournit une interface pouvant être utilisée pour dériver des vérificateurs de code octet certifiés assurant d'autres propriétés de sécurité comme la confidentialité ou la disponibilité. Des travaux

sont actuellement menés dans l'équipe pour développer une machine virtuelle défensive garantissant la non-interférence [16] (pas de fuite de valeurs sensibles vers des valeurs publiques d'un programme), et non pensons que ce travail sera utile pour dériver des vérificateurs de code octet renforcés assurant cette non-interférence. Il serait possible de mener un travail similaire pour le contrôle des ressources afin d'assurer la disponibilité de l'unité de calcul et de la mémoire.

CHAPITRE 6

Instanciation et outils d'aide à la preuve

Dans ce chapitre, nous allons détailler les preuves nécessaire à l'instanciation du modèle de vérificateur de code octet présenté au chapitre précédent. Avec les acquis de la réalisation de ces preuves, rendues fastidieuses par le nombre important et la complexité des instructions, nous allons étudier les moyens d'automatiser ces preuves et proposer des outils visant à cette automatisation. Ces outils se centreront sur une tactique en Coq facilitant le raisonnement sur les spécifications fonctionnelles ; l'utilisation de *Jakarta* pour la génération automatique de scripts de preuves à l'aide des informations nécessaires à l'abstraction ; l'utilisation d'outils de réécriture tels que Spike.

6.1 Notations de Coq

Dans cette section, nous introduisons les notations de Coq en rapport avec la preuve de lemmes ou théorèmes.

Les lemmes sont énoncés en Coq avec le mot-clé **`Lemma`** suivi du nom donné au lemme, du caractère `:` et du type du lemme. On passe ensuite dans le mode preuve de Coq, où l'on doit fournir une série de tactiques suffisant à démontrer le lemme. Dans ce mode de preuve, les hypothèses, données par leur nom et leur type, sont séparées du but à prouver par :

```
============================
```

Parmi les tactiques les plus utilisées, citons :

- **`Intros`** qui fait passer des quantificateurs ou produits du but dans les hypothèses ;
- **`Unfold`** qui déplie (expanse) dans le but la définition donnée en paramètre ;
- **`Simpl`** qui simplifie le but (par application de règles de réduction) ;
- **`Case`** qui effectue un raisonnement par cas sur le terme donné en paramètre ;
- **`Apply`** qui fait correspondre le but au lemme donné en paramètre et laisse à prouver les hypothèses du lemme ;
- **`Inversion`** qui permet de raisonner par cas sur le prédicat inductif donné en paramètre ;
- **`Reflexivity`** qui résout une égalité triviale ;

Il est également possible de composer des tactiques : avec la construction `tactic1 ; tactic2` la tactique `tactic2` sera appliquée à tous les sous-buts générés par `tactic1`. Enfin, la construction **`Try`** `tactic`, utilisée avec la composition, permet de tester la tactique `tactic` sans faire échouer la composition dans sa globalité.

6.2 Instanciation du modèle de BCV

Dans cette section, nous présentons l'instanciation faite de note modèle de vérificateur de code octet du chapitre précédent avec les machines virtuelles des chapitres 3 et 4. Nous n'entrerons cependant pas dans le détail des preuves nécessaires, une esquisse de celles-ci ainsi que les moyens de les démontrer sera donnée aux sections suivantes.

6.2.1 Machine virtuelle défensive

Définir notre machine virtuelle par l'intermédiaire du module `Defensive_VM` ne pose pas de problème particulier. Seuls cinq paramètres sont à fournir.

Les états, les contextes d'exécution et la fonction d'exécution sont ceux déjà définis lors de la construction de notre machine virtuelle défensive au chapitre 3. La fonction `getstack` correspond au champ `stack_f` de l'enregistrement représentant les états. Quant au prédicat `err_frame`, il agit par filtrage sur le type des états et discrimine les états étiquetés par `Abnormal`.

6.2.2 Machine virtuelle abstraite

Concernant la machine virtuelle abstraite cette fois, les états ne sont pas directement ceux de la machine virtuelle définie à la section 4.3. Il est nécessaire de séparer les points de programme de ces états. Le paramètre `loc` du module `Abstract_VM` devient alors :

```
Definition loc := (cap_method_idx * bytecode_idx).
```

et le paramètre `state` est défini à l'aide d'un nouveau type d'états ne comportant que pile d'opérandes et variables locales :

```
Record s_jcvm_state : Set := {
  (* operand stack *)
  s_opstack       : (list avm.valu);
  (* local variables *)
  s_locvars       : (list (option avm.valu))
}.
Inductive s_returned_state  : Set :=
  s_Normal : s_jcvm_state→s_returned_state
| s_Abnormal : s_returned_state.
Inductive state
```

On fournit cependant des fonctions de conversion de et vers le type `jcvm_state` de la machine virtuelle abstraite de la section 4.3.

La fonction `exec` fait directement appel à la fonction d'exécution de la machine virtuelle abstraite. Les états d'erreurs sont à nouveau discriminés par filtrage sur leur étiquette (`s_Abnormal`). Enfin, la fonction successeurs `succs` analyse l'instruction pointée par l'emplacement de programme et retourne :

- les branchements possibles pour les instructions de branchement ;
- l'adresse de retour contenue dans la variable visée par l'opérande pour l'instruction `ret` ;
- une liste vide pour l'instruction `return`
- l'adresse de l'instruction suivante pour les autres instructions.

Il manque encore pour satisfaire le type de module `Abstract_VM`, outre les preuves, faciles, portant sur les définitions précédentes, la définition de l'ordre sur les états de la machine virtuelle.

6.2.3 Ordre sur les types et sur les états

La vérification de code octet repose sur un ordre sur les états. Ce dernier est dérivé d'un ordre induit par la relation de sous-typage des types de la machine virtuelle abstraite.

Ordre sur les types

Afin de simplifier l'ordre sur les types, nous n'ajoutons pas aux types d'élément maximal $\top$ afin de former un treillis. À la place, l'unification de deux états normaux (`sNormal`) contenant des types incomparables ne sera pas un état normal contenant $\top$ là où les types étaient incomparables, mais un état d'erreur (`sAbnormal`). On perd ainsi le registre ou l'emplacement de la pile posant problème, mais on évite de devoir considérer $\top$ comme un type de la machine virtuelle abstraite.

Les types primitifs de la machine virtuelle sont incomparables entre eux pour l'ordre utilisé sur les types.

Parmi les types de références, la classe *Java* `Object` est la super-classe commune et le type `Null` l'élément minimal. On introduit ensuite une comparaison entre les instances de classe et les interfaces, l'unification d'une instance de classe et d'une interface pouvant être une interface si elle est implémentée par l'instance de classe considérée.

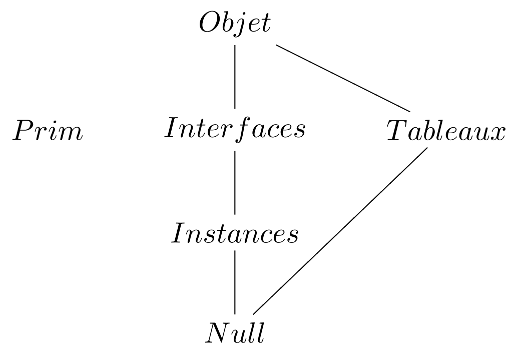

FIG. 6.1 – *Ordre sur les types de la machine virtuelle.*

Toutes les comparaisons ne sont pas indiquées sur le treillis, les instances de classes peuvent par exemple être comparées entre elles si l'une est une super-classe de l'autre, et de même pour les interfaces. Quant aux tableaux, on introduit une récursivité dans le treillis, le résultat de la comparaison de deux tableaux de types τ_1 et τ_2 étant le résultat de la comparaison de τ_1 et τ_2.

L'ordre $\prec$ sur le treillis est donné par les règles d'inférence suivantes de la figure 6.2.

Les interfaces (resp. classes) sont indicées de telle sorte que si I_m est une super-interface de I_n (resp. C_m est une super-classe de C_n) alors $m \leq_{\mathbb{N}} n$.

$$\{\rho : \texttt{type_ref}\} \vdash \rho \prec Null \text{ (Nul)}$$

$$\{\rho : \texttt{type_ref}\} \vdash Obj \prec \rho \text{ (Obj)}$$

$$\{m,n : \mathbb{N}\} \vdash I(m) \prec C(n) \text{ (ClI)}$$

$$\frac{\{m,n : \mathbb{N}\} \vdash m \leq_{\mathbb{N}} n}{I(m) \prec I(n)} \text{ (Int)}$$

$$\frac{\{m,n : \mathbb{N}\} \vdash m \leq_{\mathbb{N}} n}{C(m) \prec C(n)} \text{ (Cla)}$$

$$\frac{\{\sigma,\tau : \texttt{type}\} \vdash \sigma \prec \tau}{A(\sigma) \prec A(\tau)} \text{ (Arr)}$$

FIG. 6.2 – *Règles d'inférences de l'ordre sur les types.*

Bonne fondation de l'ordre sur les types

Tel que décrit par les règles de la figure 6.1, l'ordre $\prec$ n'est pas bien fondé. Il est en effet possible de construire une chaîne infinie décroissante pour cet ordre :

1. par l'axiome (Obj), on a $Obj \prec A[Obj]$;
2. par la règle (Arr), on déduit $A[Obj] \prec A[A[Obj]]$;
3. par transitivité, $Obj \prec A[Obj] \prec A[A[Obj]]$
4. par applications successives de la règle (Arr), on arrive à :

$$Obj \prec A[Obj] \prec A[A[Obj]] \prec A[A[A[Obj]]] \prec ...$$

On peut toutefois, en modifiant la règle (Arr), obtenir un ordre bien fondé. On définit d'abord la taille $|\sigma|$ d'un type σ comme le nombre de fois où le constructeur `Array` apparaît dans ce type. On définit ensuite la restriction $\sigma_{|m}$ d'un type σ aux m derniers constructeurs `Array` du type σ, par exemple :

$$Obj_{|2} = Obj \qquad A(A(Prim))_{|2} = A(A(Prim))$$
$$A(A(A(A(Prim))))_{|2} = A(A(Prim))$$

On remarquera que :

$$|\sigma| \leq_{\mathbb{N}} m \Leftrightarrow \sigma_{|m} = \sigma \tag{6.1}$$

L'ordre $\prec$ devient donc paramétré par un entier m et on change la règle (Arr) pour la règle :

$$\frac{\{\sigma,\tau : \texttt{type}\} \vdash \sigma \prec_m \tau \qquad |\sigma| \leq_{\mathbb{N}} m \qquad |\tau| \leq_{\mathbb{N}} m}{A(\sigma) \prec_m A(\tau)} \text{ (Arrm)}$$

Ainsi, il n'est plus possible d'obtenir une chaîne décroissante de tableaux de longueur supérieure à m.

Pour assurer la correction de $\prec_m$ vis-à-vis de $\prec$, il faut montrer que pour un programme donné, on peut borner la taille des types utilisés dans ce programme. Il suffit pour cela d'une simple analyse statique effectuant un parcours complet du programme à la recherche du plus grand type utilisé. Notons toutefois que dans le cas de *Java Card*, les tableaux de tableaux ne sont pas permis et qu'alors pour les programmes *Java Card* m vaut 1.

Ordre sur les états

À partir de l'ordre sur les types de la machine virtuelle abstraite, on construit un ordre sur les états `state` du module `Abstract_VM`.

On élève ainsi d'abord cet ordre au type `option` (pour le cas des variables locales) en un ordre $\prec_{opt}$ avec les conventions :

$$\{s : \texttt{type}\} \vdash (\texttt{Some}\ s) \prec \texttt{None} \text{ (Nul)}$$

$$\frac{\{s,t : \texttt{type}\} \vdash s \prec t}{\vdash (\texttt{Some}\ s) \prec (\texttt{Some}\ t)} \text{ (Some)}$$

Pour la pile d'opérande et les variables locales, on étend respectivement les ordres $\prec$ et $\prec_{opt}$ en $\prec_{ops}$ et $\prec_{loc}$, ordres point à point sur les listes.

Deux états `s_jcvm_state` sont alors comparables par $\prec_{sst}$ si :

$$\frac{\begin{array}{c}\{s,t : \texttt{s_jcvm_state}\} \vdash (\texttt{s_locvars}\ s) \preceq_{loc} (\texttt{s_locvars}\ t) \\ (\texttt{s_opstack}\ s) \prec_{ops} (\texttt{s_opstack}\ t)\end{array}}{s \prec_{sst} t} \text{ (Ops)}$$

$$\frac{\begin{array}{c}\{s,t : \texttt{s_jcvm_state}\} \vdash (\texttt{s_locvars}\ s) \prec_{loc} (\texttt{s_locvars}\ t) \\ (\texttt{s_opstack}\ s) \preceq_{ops} (\texttt{s_opstack}\ t)\end{array}}{s \prec_{sst} t} \text{ (Loc)}$$

où $\preceq$ est la clôture réflexive d'un ordre $\prec$.

Enfin on construit un ordre $\prec_{st}$ sur les états `state` du `Abstract_VM` :

$$\{s : \texttt{s_jcvm_state}\} \vdash \texttt{s_Abnormal} \prec_{st} s \text{ (Norm)}$$

$$\frac{\{s,t : \texttt{s_jcvm_state}\} \vdash s \prec_{sst} t}{\vdash (\texttt{s_Normal}\ s) \prec_{st} (\texttt{s_Normal}\ t)}\ \text{(Abn)}$$

6.3 Raisonnement par cas dans COQ

Dans cette section, nous allons donner un aperçu des méthodes utilisées pour la réalisation en COQ des différentes preuves portant sur les instructions des machines virtuelles. Nous verrons que ces preuves sont principalement basées sur des principes de raisonnement par cas et de réécriture. Pour illustrer les méthodes utilisées nous détaillerons pour des instructions représentatives des difficultés rencontrées, une preuve portant sur la formation des contextes d'exécution et les preuves de commutation des exécutions défensive et abstraite, de monotonie.

6.3.1 Preuve sur les contextes

Nous souhaitons ici prouver une propriété portant sur les contextes d'exécution de la machine virtuelle défensive. Cette propriété stipule que dans le cas où l'instruction à exécuter est intra-procédurale alors après exécution seul le contexte du haut de la pile a été modifié. Cela s'exprime en COQ par la formulation existentielle suivante :

```
Axiom stack_sameframe: (s:dvm.state)(f:dvm.frame)(lf:(list dvm.frame))
  (getcf s) = sameframe→
  (dvm.getstack s)=(cons f lf)→
  (EX f' | (dvm.getstack (dvm.exec s)) = (cons f' lf)).
```

Instanciée à notre machine virtuelle défensive et pour le cas particulier de l'instruction intra-procédurale `getfield`, cette propriété peut se traduire de la manière suivante :

```
Lemma GETFIELD_stack : (cap:jcprogram)(sh:sheap)(hp:heap)(f:frame)
        (lf:(list frame))(t:type)(idx:nat)(s:jcvm_state)
  let state=(Build_jcvm_state sh hp (cons f lf)) in
  (exec_instruction (getfield t idx) state cap)=(Normal s)→
  (EX f' | (stack_f s) = (cons f' lf)).
```

Procédons alors à sa démonstration.

Nous commençons par simplifier l'expression, déplier la définition de GETFIELD, introduire les différentes variables :

```
Coq < Simpl.
Coq < Intros cap sh hp f lf t idx s.
Coq < Unfold GETFIELD.
```

Nous obtenons le but :

```
cap : jcprogram
sh : sheap
hp : heap
f : frame
lf : (list frame)
t : type
idx : nat
s : jcvm_state
============================
 (Cases (head (opstack f)) of
    (Some x) ⇒
     (getfield_obj (tail (opstack f)) x t idx
       (Build_jcvm_state sh hp (cons f lf)))
  | None ⇒
     (AbortCode opstack_error (Build_jcvm_state sh hp (cons f lf)))
  end)=(Normal s)
 →(EX f':frame|(stack_f s)=(cons f' lf))
```

Il faut maintenant raisonner par cas sur `(head (opstack f))` :

```
Coq < Case (head (opstack f)).
```

On obtient ainsi deux sous-buts dont le premier est :

```
...
s : jcvm_state
============================
 (v:valu)
 (getfield_obj (tail (opstack f)) v t idx
   (Build_jcvm_state sh hp (cons f lf)))=(Normal s)
 →(EX f':frame|(stack_f s)=(cons f' lf))
```

Après introduction de la variable `v` et dépliage de `getfield_obj` :

```
Coq < Intro v.
Coq < Unfold getfield_obj.
```

```
...
s : jcvm_state
v : valu
============================
 (Cases v of
    (vPrim _) ⇒
     (AbortCode type_error (Build_jcvm_state sh hp (cons f lf)))
  | (vRef vr) ⇒
     (if (res_null vr)
      then
       (ThrowException NullPointer
         (Build_jcvm_state sh hp (cons f lf)))
      else
       Cases vr of
         vRef_null ⇒
          (AbortCode type_error (Build_jcvm_state sh hp (cons f lf)))
```

```
          | (vRef_array _ vx) ⇒
             (getfield_obj t idx (tail (opstack f)) vx
               (Build_jcvm_state sh hp (cons f lf)))
          | (vRef_instance _ vx) ⇒
             (getfield_obj t idx (tail (opstack f)) vx
               (Build_jcvm_state sh hp (cons f lf)))
          end)
     | (vNonInit _ _ _) ⇒
        (AbortCode init_error (Build_jcvm_state sh hp (cons f lf)))
     end)=(Normal s)
   →(EX f':frame|(stack_f s)=(cons f' lf))
```

Il faut à nouveau raisonner par cas, cette fois-ci sur `v`.

```
Coq < Case v.
Coq < Intro.
```

Le premier des trois sous-buts (pour `vPrim`, `vRef` et `vNonInit`) devient :

```
  ...
  s : jcvm_state
  v : valu_prim
  H : (AbortCode type_error (Build_jcvm_state sh hp (cons f lf)))
       =(Normal s)
  ============================
   (EX f':frame|(stack_f s)=(cons f' lf))
```

On obtient ici la première contradiction, réfutée par analyse des constructeurs de tête de l'égalité de l'hypothèse `H`.

```
Coq < Inversion H.
```

Le deuxième des trois sous-buts s'écrit :

```
  ...
  s : jcvm_state
  v : valu_ref
  ============================
   (if (res_null v)
    then
     (ThrowException NullPointer (Build_jcvm_state sh hp (cons f lf)))
    else
     Cases v of
       vRef_null ⇒
        (AbortCode type_error (Build_jcvm_state sh hp (cons f lf)))
     | (vRef_array _ vx) ⇒
        (getfield_obj t idx (tail (opstack f)) vx
          (Build_jcvm_state sh hp (cons f lf)))
     | (vRef_instance _ vx) ⇒
        (getfield_obj t idx (tail (opstack f)) vx
          (Build_jcvm_state sh hp (cons f lf)))
     end)=(Normal s)
   →(EX f':frame|(stack_f s)=(cons f' lf))
```

De nouveau, on procède par raisonnement par cas, sur `(res_null v)`, réfutation de la contradiction dans le cas de l'exception `NullPointer`, raisonnement par cas sur `v` et réfutation de la contradiction pour le cas `vRefNull` :

```
Coq < Case (res_null v).
Coq < Intro ; Inversion H.
Coq < Case v ; Try ( Intro ; Inversion H ; Fail ).
```

On obtient ainsi deux sous-buts à la conclusion identique :

```
...
============================
 (getfield_obj t idx (tail (opstack f)) h
   (Build_jcvm_state sh hp (cons f lf)))=(Normal s)
 →(EX f':frame|(stack_f s)=(cons f' lf))
```

Pour éviter d'avoir à prouver ces buts deux fois, on introduit un nouveau lemme exprimant ces buts :

```
Lemma getfield_obj_stack : (sh:sheap)(hp:heap)(f:frame)
  (lf:(list frame))(t:type)(idx:nat)(ops:(list valu))(hi:heap_idx)(s:jcvm_state)
  let state=(Build_jcvm_state sh hp (cons f lf)) in
  (getfield_obj t idx ops hi state)=(Normal s)→
  (EX f' | (stack_f s) = (cons f' lf)).
```

Ce lemme se prouve de manière similaire au lemme principal : raisonnement par cas, introduction de variables, dépliage de constantes, réfutation de contradiction :

```
Coq < Intros sh hp f lf t idx ops hi s.
Coq < Simpl.
Coq < Unfold getfield_obj ; Simpl.
Coq < Case (get_inst_from_heap hp hi).
Coq < Intro t1.
Coq < Case (Nth_elt (contents_i t1) idx).
Coq < Intro v.
Coq < Case v ; Try ( Intro ; Inversion H ; Fail ).
Coq < Case v ; Try ( Intro ; Inversion H ; Fail ) ;
Case t ; Try ( Intro ; Inversion H ; Fail ) ;
Case t ; Try ( Intro ; Inversion H ; Fail ) ;
Unfold res_getfield ; Simpl ;
Case (test_exception_mo f t1) ; Try ( Intros ; Inversion H ; Fail ) ;
Unfold update_frame update_opstack ; Simpl.
Coq < Case (Framework_for_Execution.le_gt_dec
        (S (length ops)) (max_opstack_size f)) ;
Coq < Intros ; Inversion H ; Simpl.
```

Ce début de script nous permet d'arriver aux conclusions non contradictoires de ce type :

```
============================
 (EX f':
    frame|(cons
```

```
              (Build_frame (cons (vPrim (vShort z)) ops) (locvars f)
                (method_loc f) (context_ref f) (S (p_count f))
                (max_opstack_size f)) lf)=(cons f' lf))
```

qui se prouve de manière triviale en fournissant explicitement la valeur de f et en montrant l'égalité :

```
Coq < Exists (Build_frame (cons (vPrim (vShort z)) ops)
                (locvars f) (method_loc f) (context_ref f)
                (S (p_count f)) (max_opstack_size f)) ;
Coq < Reflexivity.
```

Les autres conclusions non contradictoires sont de la même forme avec une valeur de type `vInt` ou `vRef` sur la pile d'opérande à la place de `vShort`.

Le sous-lemme est alors prouvé. Le lemme principal se termine facilement par application de ce sous-lemme et par réfutation des contradictions restant des précédents raisonnements par cas :

```
Coq < Apply getfield_obj_stack.
Coq < Apply getfield_obj_stack.
Coq < Intro ; Inversion H.
Coq < Intro ; Inversion H.
```

Ce que nous confirme le sybillin message :

```
Subtree proved!
```

Comme on peut le voir sur cet exemple, la preuve de cette propriété triviale sur l'instruction `getfield` s'avère particulièrement fastidieuse. Bien que cette instruction soit par nature complexe. La preuve de cette propriété sur les autres instructions s'avère en tout point comparable. On retrouve de manière systématique les mêmes schémas de preuve, principalement basés sur une analyse par cas en suivant successivement les différentes branches des instructions.

L'analogie avec les preuves portant sur des définitions inductives est flagrante. On raisonne en effet sur ces dernières en supposant pour chaque constructeur les prémisses vraies et en montrant la propriété désirée sur le terme construit. Le principe d'élimination associé à la définition inductive, accessible par la commande **`Inversion`**, permet ce mode de raisonnement. On a ainsi directement accès aux possibles termes construits. Malheureusement, aucun principe équivalent n'existe dans COQ pour raisonner de cette manière sur les fonctions. Nous avons donc été amenés à développer une tactique COQ, rendant possible un raisonnement par cas sur les fonctions avec accès direct aux extrémités des branches de la fonction (voir section 6.4).

6.3.2 Preuve de commutation

Nous illustrons ici une preuve traduisant la commutation du diagramme présenté à la figure 5.2 (aussi appelé preuve de validation croisée). Cette propriété est présente dans le type de module `mwise_struct` sous la forme :

```
Axiom diagram_sf: (a:dvm.state)
  (well_formed_memory a)→
  (getcf a) = sameframe→
  (le_avm_state (aexec (beta a)) (Snd (beta (dvm.exec a)))).
```

et peut s'écrire de manière plus simple pour le cas particulier de l'instruction `if_acmp` sous la forme :

```
Lemma IF_ACMP_commut : (cap:jcprogram)(sh:sheap)(hp:heap)(h:frame)
        (lf:(list frame))(oper:opcmp)(branch:nat)
let state=(Build_jcvm_state sh hp (cons h lf)) in
(ars2sars
  (avm.exec_instruction (if_acmp_cond oper branch) (s2as state cap) cap))=
(ars2sars
  (rs2ars (exec_instruction (if_acmp_cond oper branch) state cap) cap)).
```

où `aexec_instruction` est la fonction d'exécution abstraite, `s2as` et `rs2ars` sont les fonctions d'abstractions pour les états et états de retour respectivement (il s'agit de l'instanciation de la fonction `alpha`) et où `ars2sars` est la fonction traduisant les états de notre machine virtuelle telle que présentée à la section 4.3 vers ceux manipulés par le module `Abstract_JCVM` (sans les compteurs de programme).
On remarque que pour cette instruction particulière l'hypothèse `well_formed_memory` et l'inégalité présente dans `diagram_sf` ne sont pas nécessaires (elles le sont pour d'autres instructions, comme par exemple `getfield`).

Après introduction des différentes variables, dépliage des constantes et simplification (sic!) du terme à prouver, nous obtenons :

```
Coq < Intros cap sh hp h lf oper branch.
Coq < Unfold exec_instruction avm.exec_instruction.
Coq < Unfold IF_ACMP_COND avm.IF_ACMP_COND.
Coq < Simpl.

1 subgoal

  cap : jcprogram
  sh : sheap
  hp : heap
  h : frame
  lf : (list frame)
  oper : opcmp
  branch : nat
  ============================
```

```
(ars2sars (rs2ars
    Cases (head (opstack h)) of
      (Some v2) ⇒
       Cases (head (tail (opstack h))) of
         (Some v1) ⇒
          Cases v1 of
            (vPrim _) ⇒
             (AbortCode type_error
               (Build_jcvm_state sh hp (cons h lf)))
          | (vRef vx) ⇒
             Cases v2 of
               (vPrim _) ⇒
                (AbortCode type_error
                  (Build_jcvm_state sh hp (cons h lf)))
             | (vRef vy) ⇒
                (update_frame
                  (res_pc2
                    (res_acompare2 oper (vr2hi vx) (vr2hi vy))
                    branch h) (Build_jcvm_state sh hp (cons h lf)))
             | (vNonInit _ _ _) ⇒
                (AbortCode type_error
                  (Build_jcvm_state sh hp (cons h lf)))
             end
          | (vNonInit _ _ _) ⇒
             (AbortCode type_error
               (Build_jcvm_state sh hp (cons h lf)))
          end
       | None ⇒
          (AbortCode opstack_error
            (Build_jcvm_state sh hp (cons h lf)))
       end
    | None ⇒
       (AbortCode opstack_error (Build_jcvm_state sh hp (cons h lf)))
    end cap))
  =(ars2sars
     Cases (head (lvalu2lavalu (opstack h))) of
       (Some v2) ⇒
        Cases (head (tail (lvalu2lavalu (opstack h)))) of
          (Some v1) ⇒
           Cases v1 of
             (avm.vPrim _) ⇒
              (avm.AbortCode type_error
                (s2as (Build_jcvm_state sh hp (cons h lf)) cap))
           | (avm.vRef _) ⇒
              Cases v2 of
                (avm.vPrim _) ⇒
                 (avm.AbortCode type_error
                   (s2as (Build_jcvm_state sh hp (cons h lf)) cap))
              | (avm.vRef _) ⇒
                 (update_tframe
                   (update_tpc branch
                     (update_topstack
                       (tail (tail (lvalu2lavalu (opstack h))))
                       (s2as (Build_jcvm_state sh hp (cons h lf))
```

```
                 cap)))
              (s2as (Build_jcvm_state sh hp (cons h lf)) cap))
           | (avm.vNonInit _ _) ⇒
             (avm.AbortCode type_error
               (s2as (Build_jcvm_state sh hp (cons h lf)) cap))
           end
         | (avm.vNonInit _ _) ⇒
           (avm.AbortCode type_error
             (s2as (Build_jcvm_state sh hp (cons h lf)) cap))
         end
       | None ⇒
         (avm.AbortCode opstack_error
           (s2as (Build_jcvm_state sh hp (cons h lf)) cap))
       end
     | None ⇒
       (avm.AbortCode opstack_error
         (s2as (Build_jcvm_state sh hp (cons h lf)) cap))
     end)
```

Le terme peut paraître de taille difficilement gérable, pourtant une telle taille est fréquente dans les preuves sur la machine virtuelle. Néanmoins, seulement la première branche de part et d'autre de l'égalité nous intéresse. Nous raisonnons à nouveau par cas sur celle-ci, en conservant l'information sur la branche suivie grâce à la tactique `CaseEq` :

```
Coq < CaseEq '(head (opstack h)).
Coq < Intros v H.
```

```
  ...
  branch : nat
  v : valu
  H : (head (opstack h))=(Some valu v)
  ============================
   (ars2sars (rs2ars
       Cases (head (tail (opstack h))) of
         (Some v1) ⇒
          Cases v1 of
            ...
          end
        | None ⇒
          (AbortCode opstack_error (Build_jcvm_state sh hp (cons h lf)))
       end cap))
     =(ars2sars
        Cases (head (lvalu2lavalu (opstack h))) of
          ...
        end)
```

L'égalité n'a pas été réécrite du côté droit de l'égalité car les termes ne sont pas identiques (`lvalu2lavalu` correspond à l'abstraction d'une liste de valeurs). Mais, comme `lvalu2lavalu` est une traduction point à point de la liste (utilisation de `map`), l'existence d'un élément en tête de liste dans

`(opstack h)` garantit l'existence d'un tel élément dans la liste traduite. C'est ce qu'exprime la propriété :

```
Lemma head_val : (l:(list valu))(a:valu)
 (head l)=(Some ? a)→
 (head (lvalu2lavalu l))=(Some ? (valu2avalu a)).
```

On applique donc cette propriété et on recommence la même démarche sur la branche suivante, suivie d'un raisonnement par cas sur v1 :

```
Coq < Rewrite (head_val ? ? H).
Coq < CaseEq '(head (tail (opstack h))).
Coq < Intros v0 H1.
Coq < Rewrite (hdtl_val ? ? H1).
Coq < Case v0 ; Simpl ; Auto.
Coq < Intro v1.
Coq < Case v1 ; Simpl ; Auto.
Coq < Intro vr.
Coq < Case v ; Simpl ; Auto.
Coq < Intro v2.
Coq < Case v2 ; Simpl ; Auto.
Coq < Intro vr0.

  ============================
(ars2sars
     (rs2ars
       (update_frame
         (res_pc2 (res_acompare2 oper (vr2hi vr) (vr2hi vr0)) branch h)
         (Build_jcvm_state sh hp (cons h lf))) cap))
     =(ars2sars
        (avm.update_frame
          (avm.update_pc branch
            (avm.update_topstack (tail (tail (lvalu2lavalu (opstack h))))
              (s2as (Build_jcvm_state sh hp (cons h lf)) cap)))
          (s2as (Build_jcvm_state sh hp (cons h lf)) cap)))
```

On déplie les différentes constantes :

```
Coq < Unfold tres_pc2. Unfold update_tpc res_pc2.
Coq < Case (res_acompare2 oper (vr2hi vr) (vr2hi vr0)) ;
Unfold update_opstack update_topstack update_pc ; Simpl.

  ============================
   (ars2sars
     (rs2ars
       (update_frame
         (Build_frame (tail (tail (opstack h))) (locvars h)
           (method_loc h) (context_ref h) branch (max_opstack_size h))
         (Build_jcvm_state sh hp (cons h lf))) cap))
     =(ars2sars
        (avm.update_frame
          (avm.Build_frame (tail (tail (lvalu2lavalu (opstack h))))
            (lovalu2loavalu (locvars h)) (method_loc h) branch
            (max_opstack_size h))
          (s2as (Build_jcvm_state sh hp (cons h lf)) cap)))
```

Cette égalité se montre facilement avec les lemmes suivant sur les listes :

```
Lemma tail2_lv2lav2 : (l:(list valu))
 (lvalu2lavalu (tail (tail l))) = (tail (tail (lvalu2lavalu l))).
Lemma length_lvalu2lavalu : (l:(list valu))
(length l)=(length (lvalu2lavalu l)).
```

Il reste à montrer les cas d'erreurs :

```
  ...
  branch : nat
  v : valu
  H : (head (opstack h))=(Some valu v)
  H1 : (head (tail (opstack h)))=(None valu)
  ============================
   (ars2sars
     (rs2ars
       (AbortCode opstack_error (Build_jcvm_state sh hp (cons h lf)))
       cap))
     =(ars2sars
        Cases (head (tail (lvalu2lavalu (opstack h)))) of
          (Some v1) ⇒ ...
```

On cherche à nouveau à faire correspondre l'état des deux piles d'opérandes :

```
Lemma hdtl_err : (l:(list valu))
 (head (tail l))=(None ?)
  →(head (tail (lvalu2lavalu l)))=(None ?).

Coq < Intro H1.
Coq < Rewrite (hdtl_err ? H1) ; Simpl.

  ============================
  s_Abnormal=s_Abnormal
```

Le script se termine finalement par :

```
Coq < Reflexivity.

Coq < Intro H.
Coq < Rewrite (head_err ? H) ; Simpl.
Coq < Reflexivity.
```

Cet exemple de preuve illustre bien le schéma général des preuves portant sur la commutation des machines virtuelles : raisonnement par cas sur l'instruction défensive et corrélation des branchements suivis dans l'instruction abstraite par réécriture.

La nouveauté par rapport aux preuves décrites à la section 6.3.1 porte donc sur les étapes de réécriture. On est ainsi conduit à définir un lemme pour chacune des fonctions manipulant les structures de données de la mémoire exprimant leur comportement de morphisme vis-à-vis des fonctions d'abstraction. Pour la fonction `head`, ce sont les lemmes `head_val` et `head_err` ci-dessus.

6.3.3 Preuve de monotonie

La preuve de monotonie exprime la propriété de monotonie de l'exécution abstraite avec l'ordre sur les états abstraits. Elle apparaît dans la formulation du vérificateur de code octet par la déclaration COQ suivante :

```
Axiom exec_mon : (l:loc)(monotone ? less_state (exec l)).
```

ou bien encore en dépliant la définition de monotone sous la forme :

```
Axiom exec_mon : (l:loc)(a,a':state)
  (clos_refl_trans ? less_state a a') →
    (clos_refl_trans ? less_state (exec l a) (exec l a')).
```

La tactique de preuve déployée pour démontrer cette propriété n'est pas fondamentalement différente de celle présentée précédemment pour la validation croisée. Encore une fois, elle procède par cas, mais nécessite ici de la réécriture utilisant la prémisse de l'énoncé de monotonie. Nous allons détailler cela plus précisément en fournissant un squelette de preuve pour l'instruction if_acmp. La propriété de monotonie pour cette instruction peut s'écrire sous la forme :

```
Lemma exec_mon_if_acmp_cond :
(s,s':tjcvm_state)(op:opcmp)(b:bytecode_idx)
(lt_jcvm_state s s')→
(le_srs (rs2srs (exec_instr (if_acmp_cond op b) s))
        (rs2srs (exec_instr (if_acmp_cond op b) s'))).
```

Après dépliage des constantes et simplifications, on obtient le but :

```
  h2 : jcvm_state
  h1 : jcvm_state
  op : opcmp
  b : bytecode_idx
  H : (lt_frame h2 h1)
  ============================
   (clos_refl_trans short_rs lt_srs (rs2srs
       Cases (head (opstack h2)) of
         (Some v2) ⇒
          Cases (head (tail (opstack h2))) of
            (Some v1) ⇒
             Cases v1 of
               (vPrim _) ⇒ (AbortCode type_error h2)
             | (vRef _) ⇒
                Cases v2 of
                  (vPrim _) ⇒ (AbortCode type_error h2)
                | (vRef _) ⇒ (update_frame (res_pc2 true b h2) h2)
                | (vNonInit _ _) ⇒ (AbortCode type_error h2)
                end
             | (vNonInit _ _) ⇒ (AbortCode type_error h2)
             end
          | None ⇒ (AbortCode opstack_error h2)
```

```
        end
      | None ⇒ (AbortCode opstack_error h2)
      end) (rs2srs
      Cases (head (opstack h1)) of
        (Some v2) ⇒
         Cases (head (tail (opstack h1))) of
           (Some v1) ⇒
            Cases v1 of
              (vPrim _) ⇒ (AbortCode type_error h1)
            | (vRef _) ⇒
               Cases v2 of
                 (vPrim _) ⇒ (AbortCode type_error h1)
               | (vRef _) ⇒ (update_frame (res_pc2 true b h1) h1)
               | (vNonInit _ _) ⇒ (AbortCode type_error h1)
               end
            | (vNonInit _ _) ⇒ (AbortCode type_error h1)
            end
         | None ⇒ (AbortCode opstack_error h1)
         end
      | None ⇒ (AbortCode opstack_error h1)
      end))
```

sur lequel on commence un raisonnement par cas en conservant l'information du cas suivi qui ne simplifiera que l'exécution sur `h2` :

```
Coq < CaseEq '(head (opstack h2)) ; Intros.
```

Il faut maintenant faire correspondre le cas suivi pour l'état `h2` pour l'état `h1`. On utilise l'hypothèse `H : (lt_frame h2 h1)` car si les états `h2` et `h1` sont comparables alors leur pile d'opérandes également et l'ordre sur ces dernières implique qu'elles ont même taille. Ceci est exprimé par la propriété suivante :

```
Lemma le_lott_ex_hd : (l,l':(list (option valu)))(a:(option valu))
(le_lott l l')→(head l)=(Some ? a) →
(EX a' | (head l')=(Some ? a') ∧
         (clos_refl_trans ? less_opt_tt a a')).
```

Le but obtenu s'écrit alors :

```
  H0 : (head (opstack h2))=(Some valu t)
  x : valu
  H1 : (head (opstack h1))=(Some valu x)
  H2 : (clos_refl_trans valu lat_order t x)
  ============================
   (clos_refl_trans short_rs lt_srs (rs2srs
      Cases (head (tail (opstack h2))) of
        (Some v1) ⇒ ...
      | None ⇒ (AbortCode opstack_error h2)
      end) (rs2srs
      Cases (head (tail (opstack h1))) of
        (Some v1) ⇒ ...
      | None ⇒ (AbortCode opstack_error h1)
```

```
      end))
```

On recommence à nouveau pour `(head (tail (opstack h2)))` et l'on arrive au but :

```
  H5 : (clos_refl_trans valu lat_order t0 x0)
  ============================
   (clos_trans short_rs lt_srs (rs2srs
       Cases t0 of
         (vPrim _) ⇒ (AbortCode type_error h2)
       | (vRef _) ⇒ ...
       | (vNonInit _ _) ⇒ (AbortCode type_error h2)
       end) (rs2srs
       Cases x0 of
         (vPrim _) ⇒ (AbortCode type_error h1)
       | (vRef _) ⇒ ...
       | (vNonInit _ _) ⇒ (AbortCode type_error h1)
       end))
```

Comme toujours, on raisonne à nouveau par cas sur la première branche du but, ici sur `t0`. Cela réécrit également l'hypothèse

```
H5 : (clos_refl_trans valu lat_order (vPrim t1) x0)
```

L'ordre sur les types nous permet alors de déduire que `x0` pour être comparable à `t1` doit lui aussi être un type primitif. Et l'on obtient ainsi le but trivial :

```
  ============================
(clos_trans short_rs lt_srs sAbnormal sAbnormal)
```

Des propriétés similaires de l'ordre sur les types nous permet d'aboutir au cas intéressant pour cette instruction :

```
  ============================
(clos_trans short_rs lt_srs (rs2srs
  (update_frame
    (update_pc b (update_opstack (tail (tail (opstack h2))) h2))
   h2)) (rs2srs
   (update_frame
     (update_pc b (update_opstack (tail (tail (opstack h1))) h1))
    h1)))
```

résolu par des propriétés de monotonie sur les fonctions de mise à jour d'état et sur la fonction `tail`.

Par rapport aux preuves de validation croisée, les preuves de monotonie se présentent comme plus difficiles car on raisonne sur un ordre intrinsèquement plus complexe que la fonction d'abstraction. De manière parallèle, les lemmes utilisés pour la réécriture contiennent des quantificateurs existentiels qui empêchent une réécriture directe : il faut d'abord faire apparaître

l'élément quantifié puis accéder à la propriété de réécriture portant sur cet élément.

6.4 La tactique Coq Analyze

Nous présentons ici la tactique Coq `Analyze` développée par Pierre Courtieu et permettant un raisonnement par cas sur les fonctions exécutables [17].

Cette tactique fonctionne de manière similaire aux principes d'induction fournis par des assistants à la preuve comme Coq. Ces principes permettent de raisonner par cas et par récurrence sur les éléments d'un type inductif. Ainsi à la définition Coq du type inductif `nat`

```
Inductive nat : Set := O : nat | S : nat → nat.
```

se voit associé un principe d'induction de type :

```
nat_ind : (P:nat→Prop)(P O)→((n:nat)(P n)→(P (S n)))→(n:nat)(P n).
```

Généralement, ces principes sont générés automatiquement pour les spécifications relationnelles, mais pas pour les spécifications fonctionnelles. On préfère alors souvent les spécifications relationnelles pour le raisonnement même si les spécifications fonctionnelles présentent les avantages d'être plus naturelles à écrire et de pouvoir être exécutées et extraites vers du code.

Pour une fonction donnée, la tactique `Analyze` suit la forme de cette dernière pour construire un principe d'induction. Par exemple la fonction suivante à quatre branches

```
Fixpoint greatest_g [n:nat] : (option nat) :=
Cases n of
| O ⇒ (None ?)
| (S m) ⇒
  Cases (g n) of
  | true ⇒ (Some ? n)
  | false ⇒ Cases (h n) of
              | true  ⇒ (Some ? n)
              | false ⇒ (greatest_g m)
              end
  end
end.
```

conduit à la création du principe :

```
greatest_g_ind: (Q:(nat → Prop))
  ((n:nat) (Q O))
 → ((m:nat) (g (S m)) = true → (Q (S m)))
 → ((m:nat) (h (S m)) = true → (g (S m))=false → (Q (S m)))
```

```
→ ((m:nat) (Q m) → (h (S m)) = false → (g (S m)) = false → (Q (S m)))
→  (n:nat) (Q n)
```

Appliquée à certaines preuves de notre formalisation, comme celle présentée à la section 6.3.1 sur les contextes d'exécution, cette tactique réduit considérablement les scripts de preuves. On accède directement aux cas intéressants tout en gardant en hypothèse la trace des chemins traversés pour arriver au cas considéré.

Par exemple pour le lemme `getfield_obj_stack` le script suivant, beaucoup plus court et portable que celui de la section 6.3.1, élimine directement 34 cas contradictoires et mène aux seuls 5 cas intéressants de l'énoncé :

```
Intros sh hp f lf t idx ops hi s state.
Unfold getfield_obj.
Analyze (getfield_obj t idx ops hi state) ;
Try ( Intros ; Inversion H ; Fail ).

5 subgoals

  ...
  vnod : Z
  t0 : type_prim
  _eg_6 : t0=Byte
  _eg_5 : t=(Prim Byte)
  _eg_4 : v=(vBoolean vnod)
  _eg_3 : nod=(vPrim (vBoolean vnod))
  _eg_2 : (Nth_elt (contents_i u) idx)
            =(Some valu (vPrim (vBoolean vnod)))
  _eg_1 : (get_inst_from_heap (heap_f state) vx)=(Some type_instance u)
  ============================
   (res_getfield ops (vPrim (vShort vnod)) state u)=(Normal s)
  →(EX f':frame|(stack_f s)=(cons f' lf))

subgoal 2 is:
 (res_getfield ops (vPrim (vShort vnod)) state u)=(Normal s)
 →(EX f':frame|(stack_f s)=(cons f' lf))
subgoal 3 is:
 (res_getfield ops (vPrim (vShort z)) state u)=(Normal s)
 →(EX f':frame|(stack_f s)=(cons f' lf))
subgoal 4 is:
 (res_getfield ops (vPrim (vInt z)) state u)=(Normal s)
 →(EX f':frame|(stack_f s)=(cons f' lf))
subgoal 5 is:
 (res_getfield ops (vRef v) state u)=(Normal s)
 →(EX f':frame|(stack_f s)=(cons f' lf))
```

où les hypothèses préfixées par `_eg_` sont celles générées par la tactique `Analyze`.

Les 5 sous-buts restant sont résolus enfin de la même manière qu'à la section 6.3.1.

Ce procédé efficace pour raisonner sur des définitions fonctionnelles sera désormais intégré à la prochaine version de Coq.

6.5 Génération automatique des preuves par Spike

Spike [26] fournit un environnement pour vérifier des formules propositionnelles à l'aide de règles de réécriture conditionnelles du premier ordre, similaires à celles manipulées par *Jakarta*. Toutefois, afin de pouvoir exprimer certaines propriétés rencontrées avec les machines virtuelles, Spike a dû être étendu pour gérer les variables existentielles [27].

La méthode de preuve de Spike est basée sur des techniques comme la réécriture conditionnelle, l'analyse de cas, la subsomption, l'induction implicite. Cependant, il est nécessaire d'interagir avec le système de preuves de façon à guider, par la donnée de stratégies, l'ordre d'application des règles d'inférence, sans quoi la recherche de preuve pourrait diverger. Notons, pour le cas de la validation croisée des machines virtuelles, que des informations provenant de l'abstraction menée (comme celles utilisées par *Jakarta*, voir ci-dessous) permettraient de guider plus efficacement, et automatiquement, les stratégies de preuves utilisées par Spike. En cas d'échec de la recherche de preuve, le système fournit un contre-exemple qui peut être utilisé pour localiser l'origine de l'erreur dans les spécifications.

La réalisation des preuves de validation croisée entre les machines virtuelles offensive et défensive a été réalisée par Sorin Stratulat en Spike. Toutes les instructions, sauf 4, ont ainsi pu être traitées automatiquement. L'intervention nécessaire pour fournir des stratégies rend les bénéfices de l'application de Spike pour les preuves de monotonie et de validation croisée entre les machines virtuelles abstraite et défensive moins évidents.

Les preuves réalisées par Spike ne peuvent malheureusement pas être réutilisées par Coq, ce qui pose problème pour l'instanciation de notre modèle de vérificateur de code octet. Des travaux en cours [96] offriront à terme cette possibilité pour le système de réécriture Elan [25] .

6.6 Génération automatique des preuves par Jakarta

Nous présentons dans cette section l'utilisation de *Jakarta* pour la construction de manière automatique de preuves nécessaires à l'instanciation de notre modèle de vérificateur de code octet. L'intégration des techniques présentées ci-dessous à l'outil *Jakarta* a été réalisée par Pierre Courtieu [10].

Concernant les preuves de validation croisée des machines virtuelles, on peut remarquer que, pour une fonction d'abstraction α, une fonction f et sa version abstraite $\hat{f}$, celles-ci ont la forme :

$$\forall x_i.(Prec_{\hat{f}}\ \overrightarrow{x_i}) \ \Rightarrow\ (\mathcal{R}\ \ (\alpha(f\ \overrightarrow{x_i}))\ \ \hat{f}\ (\alpha\ \overrightarrow{x_i}))$$

où $\mathcal{R}$ est une relation binaire (par exemple l'égalité ou une relation de sous-typage) et $Prec_{\hat{f}}$ une conjonction de prémisses restreignant la conclusion aux seuls cas où elle est vraie. Dans le cas de la preuve de commutation des machines virtuelles défensive et abstraite par exemple, on ne considérera pas les cas où l'instruction défensive lance une exception puisqu'il n'y a pas d'équivalent abstrait.

Jakarta fournit un mécanisme pour générer automatiquement les prémisses du lemme, et sa preuve. Les prémisses sont calculées à partir des directives d'abstractions destinées à construire la machine virtuelle. Il apparaît important que ces prémisses soient correctes : trop restrictives elles empêcheraient de prouver la correction globale de la machine virtuelle comme une conséquence du lemme prouvé ; trop permissives elles empêcheraient de prouver le lemme lui-même.

Considérons par exemple les commandes d'abstraction suivantes :

Suppression d'une conclusion. Appliquée à une règle de la forme :

$$l_1 \rightarrow r_1 \ldots l_n \rightarrow r_n \Rightarrow (f\ \overrightarrow{x_i}) \rightarrow (...t...)$$

la commande `delete t`, destinée à enlever une règle contenant t dans la conclusion, ajoute la prémisse suivante à $Prec_{\hat{f}}$:

$$\neg(f\ \overrightarrow{x_i}) = (...t...)$$

Par exemple, dans le cas l'abstraction de types, $Prec_{\hat{f}}$ contiendra une condition de la forme $(f\ x_i) \neq (\texttt{ThrowException}\ e)$.

Suppression d'une prémisse. Appliquée à une règle de la forme :

$$l_1 \rightarrow r_1 \ldots (...t...) \rightarrow r_i \ldots l_n \rightarrow r_n \Rightarrow g \rightarrow d$$

la commande `delete t`, destinée à enlever une prémisse contenant t dans la conclusion, ajoute la prémisse suivante à $Prec_{\hat{f}}$:

$$\neg(l_1 = r_1 \wedge \cdots \wedge l_n = r_n)$$

Propagation des prémisses. De la même manière que l'abstraction d'une fonction f en $\hat{f}$ se réalise en abstrayant les fonctions auxiliaires dont f dépend, les prémisses nécessaires à établir la preuve pour ces fonctions auxiliaires sont ajoutées aux prémisses pour f. Plus précisément,

pour chaque fonction h apparaissant dans une règle (non enlevée par l'abstraction) de la forme :

$$l_1 \to r_1, \ldots, l_i \to r_i, \ (...(h\ \overrightarrow{x_j})...) \to r_{i+1}, \ \cdots \Rightarrow g \to d$$

la prémisse suivante est ajoutée à $Prec_{\hat{f}}$:

$$(l_1 = r_1 \wedge \cdots \wedge l_i = r_i) \to (Prec_{\hat{h}}\ \overrightarrow{x_j})$$

Un script de preuve Coq par défaut est donné par l'utilisateur et appliqué à chaque fonction. Il fait appel à la tactique `Analyze` décrite à la section 6.4 pour un raisonnement par cas, il recherche les contradictions dans les hypothèses et fait appel sinon à de la réécriture avec les lemmes générés pour prouver les buts restants. En cas d'échec du script par défaut (pour les cas où la preuve du fait de sa complexité ne peut être automatisée), l'utilisateur peut fournir lui-même la preuve dans le fichier Coq généré pour l'instruction. Il sera conservé si l'abstraction est réalisée à nouveau.

La démarche détaillée ci-dessus apparaît la plus efficace pour l'automatisation des preuves, principalement parce que les informations sur l'abstraction réalisée sont disponibles. Cela permet ainsi de construire automatiquement les énoncés des lemmes à prouver et des sous-lemmes dont ils dépendent avec les hypothèses nécessaires. Appliquée aux preuves de validation croisée de la section 6.3.2, cette technique a permis de décharger près de 90 % des preuves.

Notons enfin qu'une partie des mécanismes présentés ici peuvent être utilisés pour prouver d'autres types de propriétés. Concernant les invariants portant sur une machine virtuelle par exemple, *Jakarta* permet alors de générer automatiquement les énoncés des lemmes à partir d'un modèle à fournir et de placer un script de preuve par défaut également à fournir. Des résultats très concluants ont également été obtenus pour des propriétés simples comme la formation des contextes d'exécution (cf section 6.3.1).

6.7 Conclusion

Notre modèle de vérificateur de code octet nous permet de délimiter clairement les propriétés nécessaires sur les machines virtuelles sans se préoccuper des mécanismes de la vérification. Les propriétés les plus complexes à apporter concernent la validée croisée des machines virtuelles et la monotonie de la fonction exécution abstraite. Les autres preuves, l'ordre sur les états ou des invariants triviaux des machines virtuelles posent moins de difficultés.

Lorsque l'intégralité du jeu d'instructions de la machine virtuelle *Java Card* est considéré, les schémas de preuves utilisés par les autres cadres formels sur la vérification de code octet posent des problèmes de passage à l'échelle. Nous avons résolu ces problèmes par le développement d'outils visant à l'automatisation des preuves sur les machines virtuelles. Les premiers résultats obtenus par l'utilisation de ces outils pour la construction d'un vérificateur de la sûreté du typage sont particulièrement encourageants et laissent entrevoir une application facilitée de ces techniques pour la vérification d'autres propriétés de sécurité de la plate-forme.

Notre instanciation du vérificateur de code octet comporte encore quelques invariants triviaux sur la machine virtuelle à démontrer. L'achèvement récent des fonctionnalités de *Jakarta* pour la résolution de ces preuves nous permettra de compléter rapidement les preuves manquantes.

CHAPITRE 7

Conclusion

7.1 Résultats et méthodologie

Les apports de cette thèse tiennent en deux points généraux. D'une part, cette thèse apporte une formalisation exécutable et presque complète de la plate-forme *Java Card*, d'autre part, nous fournissons une méthodologie générique, et les outils associés, pour vérifier les propriétés de cette plate-forme.

7.1.1 Formalisation de la plate-forme

Cette thèse présente une formalisation, réalisée dans l'assistant de preuves Coq, de l'intégralité des instructions de la machine virtuelle *Java Card* ainsi que de l'essentiel de l'environnement d'exécution. Cette formalisation permet l'exécution des programmes et fournit un outil (le *JCVM Tools*) destiné à traduire un programme *Java Card* vers notre représentation Coq des programmes.

Pour les fonctionnalités de l'environnement d'exécution manquantes, liées aux méthodes natives de certaines APIs, nous décrivons une méthodologie pour supporter ces méthodes natives, sans toutefois proposer d'implémentation.

Enfin, nous apportons un vérificateur de code octet, qui constitue la der-

nière pièce de la plate-forme. Ce vérificateur de code octet offre les techniques de vérification les plus récentes dans le domaine. Notons qu'il provient d'une instanciation particulière de la méthodologie destinée à la vérification de la plate-forme.

7.1.2 Méthodologie pour la vérification

Afin d'assurer certaines propriétés de la plate-forme *Java Card* (celles qui restent observables par abstraction sur un domaine fini), nous proposons une méthodologie basée sur la construction de machines virtuelles offensive et abstraite selon la propriété à observer et sur la construction d'un vérificateur de code octet. Un programme dont la vérification aura réussi ne provoquera pas à l'exécution d'erreur en rapport avec la propriété observée.

Cette méthodologie est illustrée dans notre thèse autour de l'exemple de la sûreté de typage. Nous partons d'une machine virtuelle vérifiant le bon typage des valeurs (de la machine virtuelle) et construisons un vérificateur de code octet assurant qu'un programme ne provoquera pas d'erreur de typage. Le programme pourra ainsi être exécuté de manière aussi sûre sur la machine virtuelle offensive, bien que cette dernière ne réalise pas de tests sur le typage.

Ce travail a conduit à la réalisation d'outils pour faciliter et automatiser tant que possible la construction des machines virtuelles abstraite (sur laquelle repose le vérificateur de code octet) et offensive. Il a également conduit à un modèle générique de vérificateur de code octet, disponible par des interfaces de module Coq, déchargeant au maximum les preuves portant sur le vérificateur lui-même. Notons enfin que les outils réalisés (principalement *Jakarta*) peuvent aussi servir à générer automatiquement certaines preuves sur les machines virtuelles nécessaires au vérificateur de code octet.

7.1.3 Quelques chiffres

Le tableau suivant précise le nombre de lignes de code Coq pour la formalisation des différentes machines virtuelles (les machines virtuelles offensive et abstraite ont également pu être générées indépendamment par *Jakarta*) :

	Défensive	Offensive	Abstraite
Environnement	1560	760	400
Sémantique	2850	2630	1920
Total	4410	3390	2320

Avec les 2280 lignes de librairies (sur les listes, entiers, booléens) nous obtenons un total de plus de 12400 lignes.

Pour la spécification du vérificateur de code octet, 2800 lignes de spécification et de preuves sont nécessaires.

Concernant les preuves liées à l'instanciation du vérificateur de code octet, nous comptons environ 5800 lignes pour la validation croisée entre les machines défensive et abstraite, 2000 lignes pour l'équivalent défensive et offensive (quelques instructions à finir, mais les preuves générées par *Jakarta* peuvent être reprises), 3100 lignes pour la preuve de monotonie, 2680 lignes pour les preuves de bonne formation de la mémoire et des programmes et enfin 2600 lignes pour les autres preuves nécessaires à l'instanciation.

Au total, les preuves représentent donc près de 19000 lignes et la formalisation de la plate-forme dans son ensemble près de 32400 lignes.

Si la formalisation du vérificateur de code octet peut paraître si concise par rapport aux preuves nécessaires pour l'instancier, notons toutefois que ces preuves à apporter, indispensables pour la vérification, dépendent uniquement des machines virtuelles. Ainsi leur taille est directement liée au nombre d'instructions formalisées (toutes les instructions du language *Java Card* dans notre cas).

7.2 Perspectives

Les perspectives envisageables de nos travaux s'articulent autour des trois axes que constituent la plate-forme, le vérificateur de code octet et les outils pour automatiser la méthodologie proposée dans cette thèse. Le renforcement de ces axes facilitera l'étude de propriétés de sécurité de la plate-forme autres que celle du typage présentée dans ce document.

7.2.1 Plate-forme

Afin de disposer d'une plate-forme capable d'exécuter tout programme *Java Card*, il reste à fournir pour notre formalisation une implémentation Coq des méthodes natives des APIs *Java Card*. Notons toutefois que ces méthodes ne pourront être utilisées que pour l'expérimentation car elles devront être réécrites spécifiquement pour la puce sur laquelle la machine virtuelle fonctionnera.

Nous pouvons également envisager d'intégrer la phase de liaison, actuellement réalisée par le *JCVM Tools*, à notre formalisation pour permettre d'utiliser directement des programmes *Java Card* sous leur forme de fichier

capfile. La solution la plus simple serait alors de se baser sur les travaux de Denney et Jensen [51].

Enfin, *Java* et *Java Card* partageant une partie conséquente de leurs instructions, il apparaît intéressant de proposer une formalisation, sur les mêmes principes que celle réalisée dans ce document, pour le langage *Java*. Néanmoins, pour être entièrement fonctionnelle cette formalisation nécessiterait également de considérer les dispositifs de chargement dynamique de classes, de ramasse-miettes et de fils d'exécution. Même si des travaux existent déjà sur ces thèmes, l'effort d'intégration serait non négligeable.

7.2.2 Vérificateur de bytecode

Les améliorations à apporter aux modèles de vérificateurs de code octet décrits dans cette thèse concernent principalement des optimisations de l'algorithme. Une des optimisations possible consiste à ne plus pratiquer la phase d'itération décrite dans l'algorithme sur tous les points de programmes, mais à marquer les états sur lesquels il est nécessaire d'effectuer une nouvelle passe d'exécution.

Concernant le vérificateur de code octet léger, on peut proposer un niveau de raffinement supplémentaire du modèle. Cela permettrait par exemple de réduire la taille du certificat à transmettre avec le programme.

Enfin, sur les interfaces disponibles pour accéder aux vérificateurs de code octet (telles que les analyses monovariante et polyvariante), on peut proposer par le même modèle d'autre type d'analyses hybrides, optimisant le nombre d'états à conserver dans la structure d'historique selon un objectif fixé.

7.2.3 Outils

Si la construction des machines abstraite et offensive par *Jakarta* semble aboutie dans ce qu'il est possible d'automatiser, il reste cependant certainement des améliorations à effectuer dans la génération de scripts de preuves. La connexion avec un système de réécriture, telle que l'expérimentation menée actuellement avec Elan [96], permettrait d'optimiser les procédures de réécriture que nous utilisons et de simplifier l'écriture dans *Jakarta* de scripts de preuves adaptés à la résolution d'une classe de problème donnée. Nous pensons que la puissance d'un tel outil permettrait l'automatisation presque complète des preuves des invariants sur les machines virtuelles, relativement nombreux dans l'instanciation du modèle de vérificateur de code octet mais néanmoins quasiment triviaux dans leur énoncé.

7.2.4 Sécurité

Si la sûreté du typage constitue une composante essentielle dans la sécurité de la plate-forme *Java Card*, d'autres types de propriétés de sécurité peuvent être étudiées et apporter une confiance accrue dans la plate-forme. Parmi celles accessibles aisément figurent la vérification de l'initialisation des objets, notre machine virtuelle défensive intégrant déjà les tests nécessaires. L'étude de la non-interférence (*i.e* l'absence de fuite d'informations sensibles vers des données publiques) se prêterait également très bien à la méthodologie que nous avons présentée. En effet, l'observation de la non-interférence repose aussi sur un système de types pour les valeurs. Nous pensons finalement notre méthodologie s'appliquerait à d'autres propriétés telles que le contrôle des ressources (disponibilité de la mémoire et de l'unité de calcul) ou la détection des pointeurs nuls mais nous n'avons pas encore étudiés ces directions.

Bibliographie

[1] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] Ross Anderson. Why Cryptosystems Fail, from Communications of the ACM, November, 1994. Dans *William Stallings, Practical Cryptography for Data Internetworks*. IEEE Computer Society Press, 1996.

[3] Lennart Augustsson. « Cayenne: A language with dependent types ». Dans *Proceedings of ICFP'98*, pages 239–250. ACM Press, 1998.

[4] Franz Baader et Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[5] BALI Project. `http://isabelle.informatik.tu-muenchen.de/Bali/`.

[6] Bandera project. `http://bandera.projects.cis.ksu.edu/`.

[7] Henk Barendregt. *The Lambda Calculus, its Syntax and Semantics*, volume 103 de *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[8] Henk Barendregt. Lambda calculi with types. Dans *Handbook of Logic in Computer Science*, volume 2, pages 117–310. Oxford University Press, 1992.

[9] Bruno Barras. « Verification of the Interface of a Small Proof System in Coq ». Dans E. Gimenez et C. Paulin-Mohring, éditeurs, *Proceedings of the 1996 Workshop on Types for Proofs and Programs*, volume 1512 de *Lecture Notes in Computer Science*, pages 28–45, Aussois, France, décembre 1996. Springer-Verlag.

[10] G. Barthe, P. Courtieu, G. Dufay, et S. Melo de Sousa. « Tool-

Assisted Specification and Verification of the JavaCard Platform ». Manuscrit. Version journal de [11] à soumettre.

[11] G. Barthe, P. Courtieu, G. Dufay, et S. Melo de Sousa. « Tool-Assisted Specification and Verification of the JavaCard Platform ». Dans H. Kirchner et C. Ringessein, éditeurs, *Proceedings of AMAST'02*, volume 2422 de *Lecture Notes in Computer Science*, pages 41–59. Springer-Verlag, 2002.

[12] G. Barthe, G. Dufay, M. Huisman, et S. Melo de Sousa. « Jakarta: a toolset to reason about the JavaCard platform ». Dans I. Attali et T. Jensen, éditeurs, *Proceedings of e-SMART'01*, volume 2140 de *Lecture Notes in Computer Science*, pages 2–18. Springer-Verlag, 2001.

[13] G. Barthe, G. Dufay, L. Jakubiec, et S. Melo de Sousa. « A formal correspondence between offensive and defensive JavaCard virtual machines ». Dans A. Cortesi, éditeur, *Proceedings of VMCAI'02*, volume 2294 de *Lecture Notes in Computer Science*, pages 32–45. Springer-Verlag, 2002.

[14] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, et S. Melo de Sousa. « A Formal Executable Semantics of the JavaCard Platform ». Dans D. Sands, éditeur, *Proceedings of ESOP'01*, volume 2028 de *Lecture Notes in Computer Science*, pages 302–319. Springer-Verlag, 2001.

[15] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, S. Melo de Sousa, et S.-W. Yu. « Formalisation of the Java Card Virtual Machine in Coq ». Dans S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, et A. Poetzsch-Heffter, éditeurs, *Proceedings of FTfJP'00—ECOOP Workshop on Formal Techniques for Java Programs*, 2000.

[16] Gilles Barthe, Amitabh Basu, et Tamara Rezk. « Security Types preserving Compilation ». Submitted to VMCAI'04, 2003.

[17] Gilles Barthe et Pierre Courtieu. « Efficient Reasoning about Executable Specifications in Coq ». Dans V. Carreño, C. Muñoz, et S. Tahar, éditeurs, *Proceedings of TPHOLs'02*, volume 2410 de *Lecture Notes in Computer Science*, pages 31–46. Springer-Verlag, 2002.

[18] D. Basin, S. Friedrich, M. Gawkowski, et J. Posegga. « Bytecode Model Checking: An Experimental Analysis », 2002.

[19] Patrick Behm, Paul Benoit, Alain Faivre, et Jean-Marc Meynadier. « METEOR : A successful application of B in a large project ». Dans *Proceedings of FM'99: World Congress on Formal Methods*, pages 369–387, 1999.

[20] Jan Bergstra et Jan Willern Klop. « Conditional rewrite rules: confluence and termination ». *Journal of Computer and System Sciences*, 32:323–362, 1986.

[21] Yves BERTOT. « Formalizing a JVML Verifier for Initialization in a Theorem Prover ». Dans G. BERRY, H. COMON, et A. FINKEL, éditeurs, *Proceedings of Computer Aided Verification (CAV'01)*, volume 2102 de *Lecture Notes in Computer Science*, pages 14–24. Springer-Verlag, 2001.

[22] Yves BERTOT et Pierre CASTÉRAN. « Le Coq'Art ». `http://www-sop.inria.fr/lemme/Yves.Bertot/coqart.html`. À paraître., 2003.

[23] M. BEZEM, J. W. KLOP, et R. de VRIJER, éditeurs. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2002.

[24] Dines BJØRNER et Cliff JONES. *The Vienna Development Method: The Meta-Language*, volume 61 de *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[25] P. BOROVANSKÝ, C. KIRCHNER, H. KIRCHNER, P.-E. MOREAU, et C. RINGEISSEN. « An Overview of ELAN ». Dans Claude KIRCHNER et Hélène KIRCHNER, éditeurs, *Proceedings of the International Workshop on Rewriting Logic and its Applications*, volume 15 de *Electronic Notes in Theoretical Computer Science*, Pont-à-Mousson, France, septembre 1998. Elsevier Science.

[26] A. BOUHOULA, E. KOUNALIS, et M. RUSINOWITCH. « SPIKE, an Automatic Theorem Prover ». Dans A. VORONKOV, éditeur, *Proceedings of the International Conference on Logic Programming and Automated Reasoning (LPAR'92)*, volume 624 de *Lecture Notes in Artificial Intelligence*, pages 460–462. Springer-Verlag, juillet 1992.

[27] Adel BOUHOULA. « Using induction and rewriting to verify and complete parameterized specification ». *Theorical Computer Science*, 170(1–2):245–276, 1996.

[28] Jonathan BOWEN. « Web site on Formal Methods ». `http://www.afm.sbu.ac.uk/`.

[29] Jonathan BOWEN et Victorian STAVRIDOU. « Safety-Critical Systems, Formal Methods and Standards ». *IEE/BCS Software Engineering Journal*, 8(4):189–209, juillet 1993.

[30] Lilian BURDY. « B vs. Coq to prove a garbage collector ». Dans R. J. BOULTON AND P. B. JACKSON, éditeur, *Theorem Proving in Higher Order Logics (TPHOL)*, numéro EDI-INF-RR-00?? dans Technical Report. Edinburgh University, 2001.

[31] Lilian BURDY. « JACK: Java Applet Correctness Kit », 2002. Gemplus Developer Conference.

[32] Lilian BURDY, Yoonsik CHEON, David COK, Michael D. ERNST, Joe KINIRY, Gary T. LEAVENS, K. RUSTAN, M. LEINO, et Erik POLL. « An overview of JML tools and applications ». Dans Thomas ARTS et Wan

FOKKINK, éditeurs, *Electronic Notes in Theoretical Computer Science*, volume 80. Elsevier, 2003.

[33] Ludovic CASSET. « *Construction correcte de logiciels pour carte à puce* ». Thèse de doctorat, Université d'Aix-Marseille II, 2002.

[34] Ludovic CASSET. « Development of an Embedded Verifier for Java Card Byte Code using Formal Methods ». Dans Lars-Henrik ERIKSSON et Peter Alexander LINDSAY, éditeurs, *Formal Methods – Getting IT Right*, volume 2391 de *Lecture Notes in Computer Science*, pages 290–309. Springer-Verlag, juillet 2002.

[35] Ludovic CASSET, Lilian BURDY, et Antoine REQUET. « Formal Development of an Embedded Verifier for JavaCard ByteCode ». Dans *Proceedings of DSN'02*. IEEE Computer Society, 2002.

[36] Ludovic CASSET et Jean-Louis LANET. « A Formal Specification of the Java Byte Code Semantics using the B Method ». Dans B. JACOBS, G. T. LEAVENS, P. MÜLLER, et A. POETZSCH-HEFFTER, éditeurs, *Proceedings of Formal Techniques for Java Programs*. Rapport Technique 251, Fernuniversität Hagen, 1999.

[37] Jacek CHRZĄSZCZ. « Implementation of modules in the system Coq ». Manuscript, 2003.

[38] Alonzo CHURCH. « A set of postulates for the foundation of logic ». *Annals of Mathematics 2nd series*, pages 839–864, 1933.

[39] Edmund CLARKE et Jeanette WING. « Formal Methods: State of the Art and Future Directions ». *ACM Computing Surveys*, 28(4):626–643, décembre 1996.

[40] CLDC AND THE K VIRTUAL MACHINE (KVM). `http://java.sun.com/products/cldc`.

[41] Alessandro COGLIO. « Improving the Official Specification of Java Bytecode Verification ». Dans *Proceedings of the 3rd ECOOP Workshop on Formal Techniques for Java Programs*, juin 2001.

[42] Alessandro COGLIO. « Simple Verification Technique for Complex Java Bytecode Subroutines ». Dans *Proceedings of the 4th ECOOP Workshop on Formal Techniques for Java-like Programs*, juin 2002.

[43] Alessandro COGLIO, Allen GOLDBERG, et Zhenyu QIAN. « Towards a Provably-Correct Implementation of the JVM Bytecode Verifier ». Dans *Proceedings of OOPSLA'98 Workshop on Formal Underpinnings of Java*, octobre 1998.

[44] Alessandro COGLIO, Allen GOLDBERG, et Zhenyu QIAN. « A formal specification of Java class loading ». *ACM SIGPLAN Notices*, 35(10):325–336, 2000.

[45] Richard M. COHEN. « Defensive Java Virtual Machine Specification Version 0.5 ». Manuscript, 1997.

[46] « Common Criteria ». `http://www.commmoncriteria.org/`.

[47] Thierry COQUAND, Randy POLLACK, et Makoto TAKEYAMA. « Modules as dependently typed records ». Manuscript, 2002.

[48] Judicaël COURANT. « *Un calcul de modules pour les systèmes de types purs* ». Thèse de doctorat, Ecole Normale Supérieure de Lyon, 1998.

[49] Patrick COUSOT. «Abstract interpretation». *ACM Computing Surveys (CSUR)*, 28(2):324–328, 1996.

[50] Marc DAUMAS, Laurence RIDEAU, et Laurent THÉRY. « A Generic Library for Floating-Point Numbers and Its Application to Exact Computing ». *Lecture Notes in Computer Science*, 2152:169+, 2001.

[51] Ewen DENNEY et Thomas JENSEN. « Correctness of Java Card method lookup via logical relations ». *Theoretical Computer Science*, 283:305–331, 2002.

[52] Damien DEVILLE, Antoine GALLAND, Gilles GRIMAUD, et Sebastien JEAN. « Smart Card Operating Systems: Past, Present and Future ». Dans *the 5th USENIX/NordU Conference*, Vasteras, Sweden, février 2003.

[53] R. Kent DYBVIG. *The Scheme Programming Language*. Prentice Hall, second édition, 1996.

[54] Marc ELUARD et Thomas JENSEN. « Secure object flow analysis for Java Card ». Dans *Proceedings of Cardis'02*. USENIX, 2002.

[55] Marc ELUARD, Thomas JENSEN, et Igor SIVERONI. « A Formal Specification of the Java Card Firewall ». Manuscrit, octobre 2001.

[56] Stephen N. FREUND. « The costs and benefits of java bytecode subroutines », 1998.

[57] Stephen N. FREUND et John C. MITCHELL. « The type system for object initialization in the Java bytecode language ». *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.

[58] Stephen N. FREUND et John C. MITCHELL. « A type system for the Java bytecode language and verifier ». *Journal of Automated Reasoning*, 2003. À paraître.

[59] Eduardo GIMÉNEZ et Olivier LY. « Formal Modeling and Verification of the Java Card Security Architecture: from Static Checkings to Embedded Applet Execution », 2002. Verificard'02, Marseille, 7-9 January 2002. http://www-sop.inria.fr/lemme/verificard/2002/programme.html.

[60] Gilles GRIMAUD. « *CAMILLE, un système d'exploitation ouvert pour cartes à microprocesseur* ». Thèse de doctorat, Laboratoire d'Informatique Fondamentale de Lille, décembre 2000.

[61] Yuri GUREVICH. Evolving Algebras 1993: Lipari Guide. Dans E. BÖRGER, éditeur, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[62] Chris Hankin et Igor Siveroni. « A Proposal for the JCVMLe Operational Semantic ». Manuscrit, octobre 2001.

[63] Peter H. Hartel et Luc A. V. Moreau. « Formalizing the Safety of Java, the Java Virtual Machine and Java Card ». *ACM Computing Surveys*, 33(4):517–558, décembre 2001.

[64] John Hatcliff et Matthew Dwyer. « Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software ». Dans K. G. Larsen et M. Nielsen, éditeurs, *Proceedings of CONCUR'01*, volume 2154 de *Lecture Notes in Computer Science*, pages 39–58. Springer-Verlag, 2001.

[65] Ludovic Henrio et Bernard Paul Serpette. « A Parametrized Polyvariant Bytecode Verifier ». Dans *Actes des journées JFLA*, Chamrousse, France, janvier 2003.

[66] Jim Huggins. « Web site on Abstract State Machine ». `http://www.eecs.umich.edu/gasm/`.

[67] International Organization for Standardization. « International Standard ISO/IEC 7816: Integrated circuit cards with contact parts », 1987.

[68] Bart Jacobs et Erik Poll. « Coalgebras and Monads in the Semantics of Java ». *Theoretical Computer Science*, 291(3):329–349, 2003.

[69] « Java Card Forum ». `http://www.javacardforum.org/`.

[70] Thomas Jensen, Daniel Le Métayer, et Tommy Thorn. « Security and Dynamic Class Loading in Java: A Formalisation ». Dans *IEEE International Conference on Computer Languages*, pages 4–15, Chicago, Illinois, 1998.

[71] « JML Specification Language ». `http://www.jmlspecs.org`.

[72] R. Juellig, Y. Srinivas, et J. Liu. « SPECWARE: An Advanced Environment for the Formal Development of Complex Software Systems ». Dans *Proceedings of AMAST'96*, volume 1101 de *Lecture Notes in Computer Science*, pages 551–554. Springer-Verlag, 1996.

[73] R. Jullig, Y. V. Srinivas, L. Blaine, L.-M. Gilham, A. Goldberg, C. Green, J. Mc-Donald, et R. Waldinger. « Specware language manual ». Rapport Technique, Krestel Institute, 1995.

[74] Matt Kaufmann et J Strother Moore. « ACL2: An Industrial Strength Version of Nqthm ». Dans *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 23, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.

[75] Gary Kildall. « A unified approach to global program optimization ». Dans *Conference Record of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 194–206. ACM Press, janvier 1973.

[76] Gerwin Klein. « *Verified Java Bytecode Verification* ». PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[77] Gerwin Klein et Tobias Nipkow. « Verified Lightweight Bytecode Verification ». *Concurrency and Computation: Practice and Experience*, 13(13):1133–1151, 2001.

[78] Gerwin Klein et Tobias Nipkow. « Verified Bytecode Verifiers ». *Theoretical Computer Science*, 298:583–626, 2003.

[79] Gerwin Klein et Martin Strecker. « Verified Bytecode Verification and Type-Certifying Compilation ». *Journal of Logic Programming*, 2002. À paraître.

[80] Skander Kort. « Java Card Common Criteria Using Coq ». Manuscrit, avril 2001.

[81] Jean-Louis Lanet et Antoine Requet. « Formal proof of smart card applets correctness ». Dans J.-J. Quisquater et B. Schneier, éditeurs, *Proceedings of the Third Smart Card Research and Advanced Application Conference (CARDIS'98)*, volume 1820 de *Lecture Notes in Computer Science*, pages 85–97. Springer-Verlag, septembre 1998.

[82] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, et J. Vouillon. « *The Objective Caml system, release 3.06. Documentation and user's manual* », 2002.

[83] Xavier Leroy. « On-card Bytecode Verification for Java Card ». Dans I. Attali et T. Jensen, éditeurs, *Proceedings e-Smart 2001*, volume 2140, pages 150–164. Springer-Verlag, 2001.

[84] Xavier Leroy. « Java bytecode verification: algorithms and formalizations ». *Journal of Automated Reasoning*, 2003. À paraître dans Special Issue on Java Bytecode Verification.

[85] Tim Linhold et Frank Yellin. « *The Java Virtual Machine Specification* ». Sun Microsystems, Inc., Palo Alto/CA, USA, second édition, 1999.

[86] LOOP project. http://www.cs.kun.nl/ita/research/projects/loop/.

[87] Maosco Ltd. « MULTOS ». Documents fondateurs et bulletins techniques disponibles sur http://www.multos.com/.

[88] Claude Marché, Christine Paulin, et Xavier Urbain. « The Krakatoa Tool for JML/Java Program Certification ». Disponible à http://krakatoa.lri.fr, 2003.

[89] Coq Development Team. « The Coq Proof Assistant Reference Manual – Version V7.4 ». http://coq.inria.fr/doc/main.html. À paraître., 2003.

[90] Gary McGraw et Edward Felten. *Securing Java: getting down to business with mobile code*. John Wiley and Sons, Inc., New York, USA, 1999.

[91] Henk Meijer et Erik Poll. « Towards a full formal specification of the Java Card ». Dans I. Attali et T. Jensen, éditeurs, *Smart Card*

Programming and Security (Proceedings of e-Smart'01), numéro 2140 dans Lecture Notes in Computer Science, pages 165–178. Springer-Verlag, septembre 2001.

[92] Simão MELO DE SOUSA. « *Outils et techniques pour la vérification formelle de la plate-forme JavaCard* ». Thèse de doctorat, Université de Nice Sophia-Antipolis, 2003.

[93] J. MEYER, P. MÜLLER, et A. POETZSCH-HEFFTER. « The JIVE System—Implementation Description ». `http://softech.informatik.uni-kl.de/en/publications/jive.html`, 2000.

[94] MICROSOFT CORPORATION. « Smart Cards for Windows ». `http://www.microsoft.com/smartcards/`.

[95] George C. NECULA. « Proof-Carrying Code ». Dans *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, janvier 1997.

[96] Quang Huy NGUYEN, Claude KIRCHNER, et Helene KIRCHNER. « External rewriting for skeptical proof assistants ». *Journal of Automated Reasonnng*, 29(3–4):309–336, 2002.

[97] Tobbias NIPKOW et David VON OHEIMB. « Java_{light} is Type-Safe — Definitely ». Dans *Proceedings of POPL'98*, pages 161–170. ACM Press, 1998.

[98] Lawrence C. PAULSON. « The Isabelle Reference Manual ». Rapport Technique 283, The University of Cambridge Computer Laboratory, février 2001. Disponible par ftp `ftp.cl.cam.ac.uk`, avec l'implementation.

[99] Joachim POSEGGA et Harald VOGT. « Java bytecode verification using model checking ». Dans *Proceedings of the OOPSLA'98 Workshop on Formal Underpinnings of Java*, octobre 1998.

[100] Cornelia PUSCH. « Proving the Soundness of a Java Bytecode Verifier Specification in Isabelle/HOL ». Dans W. R. CLEAVELAND, éditeur, *Proceedings of TACAS'99*, volume 1579 de *Lecture Notes in Computer Science*, pages 89–103. Springer-Verlag, 1999.

[101] Zhenyu QIAN. A Formal Specification of Java™ Virtual Machine Instructions for Objects, Methods and Subroutines. Dans Jim ALVES-FOSS, éditeur, *Formal Syntax and Semantics of Java™*, volume 1523 de *Lecture Notes in Computer Science*, pages 271–312. Springer-Verlag, 1999.

[102] Zhenyu QIAN. « Standard fixpoint iteration for Java bytecode verification ». *ACM Transactions on Programming Languages and Systems*, 22(4):638–672, 2000.

[103] Antoine REQUET. « A B model for ensuring soundness of a large subset of the Java Card virtual machine ». *Science of Computer Programming*, pages 283–306, 2003.

[104] Eva Rose. « Lightweight bytecode verification ». *Journal of Automated Reasoning*, 2003. À paraître.

[105] John Rushby. « Theorem Proving for Verification ». Dans Franck Cassez, éditeur, *Proceedings of MOVEP'2k: Modelling and Verification of Parallel Processes*. Springer-Verlag, juin 2000.

[106] K. Rustan, M. Leino, Greg Nelson, et James B. Saxe. « ESC/-Java User's Manual ». Rapport Technique, Compaq Systems Research Center, 2000. Technical Note 2000-002.

[107] Donald Sannella. A Survey of Formal Software Development Methods. Dans Richard Thayer et Andrew McGettrick, éditeurs, *Software Engineering: A European Perspective*, pages 281–297. IEEE Computer Society Press, 1993.

[108] Schlumberger. « .NET Card ». `http://www.smartcards.net/infosec/.NET.html`.

[109] Joachim Schmid. « Web site on AsmGofer ». `http://www.tydo.de/AsmGofer/`.

[110] N. Shankar, S. Owre, et J.M. Rushby. « *The PVS Proof Checker: A Reference Manual* ». Computer Science Laboratory, SRI International, février 1993. Supplemented with the PVS2 Quick Reference Manual, 1997.

[111] Igor Siveroni. « Formalisation of the Java Card Runtime Environment and API ». Manuscrit, novembre 2002.

[112] Igor Siveroni. « Operational Semantics of the Java Card Virtual machine ». Manuscrit, novembre 2002.

[113] Michael Spivey. « An Introduction to Z and Formal Specification ». *IEEE Software Engineering Journal*, 4(1):40–50, 1989.

[114] Robert Stärk, Joachim Schmid, et Egon Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer-Verlag, 2001.

[115] Raymie Stata et Martín Abadi. « A Type System for Java Bytecode Subroutines ». Dans *Record of the ACM Symposium on Principles of Programming Languages (POPL 98)*, pages 149–160, New York, NY, 1998.

[116] Leon Sterling et Ehud Shapiro. *The Art of Prolog*. MIT Press, second édition, 1994.

[117] Martin Strecker. « Formal Verification of a Java Compiler in Isabelle ». Dans *Proceedings Conference on Automated Deduction (CADE)*, volume 2392 de *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2002.

[118] J. Strother Moore, R. Krug, H. Liu, et G. Porter. « Formal Models of Java at the JVM Level – A Survey from the ACL2 Perspective ». Dans S. Drossopoulou, éditeur, *Proceedings of Formal Techniques for Java Programs*, juin 2001.

[119] Sun Microsystems, Inc., Palo Alto/CA, USA. « *Java Card 2.2 Application Programming Interface (API)* », 2002.

[120] Sun Microsystems, Inc., Palo Alto/CA, USA. « *Java Card 2.2 Runtime Environment (JCRE) Specification* », 2002.

[121] Sun Microsystems, Inc., Palo Alto/CA, USA. « *Java Card 2.2 Virtual Machine Specification* », 2002.

[122] Trusted Logic. « Off-card bytecode verifier for Java Card », 2001. Distributed as part of Sun's Java Card Development Kit.

[123] Alan M. Turing. « On Computable Numbers, with an Application to the Entscheidungsproblem ». *Proceedings of the London Mathematical Society*, Series 2(42):230–265, 1936-1937.

[124] Joachim van den Berg et Bart Jacobs. « The LOOP Compiler for Java and JML ». Dans T. Margaria et W. Yi, éditeurs, *Proceedings of TACAS'01*, volume 2031 de *Lecture Notes in Computer Science*, pages 299–312, 2001.

[125] David von Oheimb. « *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic* ». PhD thesis, Technische Universität München, 2001.

[126] Frank Yellin. « Low level security in Java ». Dans *Proceedings of the Fourth International World Wide Web Conference*, pages 369–379. O'Reilly, 1995.

Index

`()`, 40
`<:`, 107
`[]`, 40
état
    sûr, 121
_, 41

abstraite, 74
algorithme de Kildall, 100
APDU, 24
API, 67
applette, 21
`Apply`, 130
`AVM`, 113

`bad`, 110
`BCV`, 110
`BCV_dexec`, 122
bytecode, 23, 44

CAP, 23
`Case`, 41, 130
certificat, 104
`check`, 110
classe, 45
`clos_refl_trans`, 108
contexte d'exécution, 49
conversion, 62, 65

défensive, 74
`dec_pred`, 108
`dec_type`, 108
`Definition`, 41
`down_closed`, 108
`DVM`, 118

enregistrement, 40
environnement d'exécution, 24, 47
erreurs, 59
état, 50
état de retour, 51
exécution, 61
exceptions, 59
    récupération, 60

fichier
    cap, 62
    class, 62
foncteur, 107
fonction d'abstraction, 118
`frame`, 50

`heap`, 49

`Inductive`, 40
initialisation, 105
`Instruction`, 44
instruction, 44, 51
interface, 44
intra-procédural, 118
`Intros`, 130
`Inversion`, 130

Jakarta, 86
JCVM Tools, 62

`jcvm_state`, 51
JSL, 86

`LBCV`, 126
`Lemma`, 129
`locvars`, 50

méthode, 43
  native, 68
`Method_Comp_Struct`, 118
`Module`, 107
module
  signature, 107
`Module Type`, 107
modules, 106
`monotone`, 108
monotonie, 145

`nat`, 40
noms qualifiés, 106

`Object`, 46
objets, 48
offensive, 73, 74
`opstack`, 50
`option`, 41
`OVM`, 123

paquettage, 46
pare-feu, 25
partialité, 41
pile d'opérande, 49
pile de contexte, 50
programme, 46

`reaches`, 108
`Record`, 40
`Reflexivity`, 130
registre, 49
`returned_state`, 51

script, 95
`sheap`, 50
`Simpl`, 130
sous-routine, 101
`stack`, 50
`stackmap`, 115
structure d'historique, 113
structure de point fixe, 110
système de transition
  avec erreur, 109

tas, 49
tas statique, 50
transaction, 24
transient, 24
`type`, 43
`type_prim`, 42
`type_ref`, 43
types
  primitifs, 42
  référence, 42

`Unfold`, 130
unification, 101
`update_frame`, 53
`update_opstack`, 53

vérificateur de code octet, 110, 112
  léger, 104, 126
  monovariant, 103, 112
  polyvariant, 103, 112
valeurs, 47
validation croisée, 98, 140
`valu`, 48
`valu_prim`, 48
`valu_prim`, 48
variable locale, 49

zone des constantes, 64

Acronymes

AID Applet IDentifier. Identificateur, sous forme numérique, d'applette.

APDU Application Protocol Data Unit. Format d'échange de données entre la carte à puce et le CAD.

API Application Programmer Interface. Ensemble de fonctions permettant d'accéder à des librairies.

ASCII American Standard Code for Information Interchange. Jeu de caractères minimaliste et très répandu.

ASM Abstract State Machine. Machines à états abstraits.

BCV ByteCode Verifier. Vérificateur de code octet.

CAD Card Acceptance Device. Terminal de communication avec une carte à puce.

CAP Converted APplet. Format de représentation d'un programme *Java Card*.

EAL Evaluation Assurance Level. Niveau de confiance attribué à un système par les Critères Communs.

EEPROM Electrically Erasable Read Only Memory. Mémoire à contenu modifiable et non-volatile.

GIE Groupement d'Intérêts Economiques. Partenariat entre plusieurs industriel autour d'un projet commun.

GSM Global System for Mobile communication. GSM est le standard de la téléphonie mobile en Europe.

JCVM Java Card Virtual Machine. Machine virtuelle *Java Card*.

JCRE Java Card Runtime Environment. Environnement d'exécution *Java Card*.

JVM Java Virtual Machine. Machine virtuelle *Java*.

PIN Personal Identification Number. Code d'identification personnel protégeant l'utilisation d'un service.

RAM Random Access Memory. Mémoire à contenu modifiable et volatile.

ROM Read Only Memory. Mémoire à contenu non modifiable.

SIM Subscriber Identification Module. Carte à puce assurant de nombreux services dans les téléphones GSM.

Résumé

Dans cette thèse, nous présentons une formalisation, réalisée dans l'assistant de preuve COQ, de la plate-forme *Java Card*. Cet environnement de programmation dérivé de *Java* est destiné aux cartes à puce intelligentes et à leurs impératifs particuliers de sécurité. Plus précisément, nous décrivons la formalisation de la machine virtuelle *Java Card* et d'une partie conséquente de son environnement d'exécution. Nous explicitons ensuite comment, à partir de cette machine virtuelle vérifiant dynamiquement la sûreté du typage, obtenir un vérificateur de code octet (BCV) et une machine virtuelle plus efficace à l'exécution, mais aussi sûre. Nous apportons pour le BCV un cadre générique, exprimé sous forme de modules, appliquant les techniques les plus récentes de ce domaine et simplifiant les preuves nécessaires pour assurer la correction du BCV construit. Nous montrons enfin comment la méthodologie appliquée pour la sûreté du typage peut être généralisée et décrivons les outils réalisés pour automatiser cette tâche.

Abstract

In this thesis, we present a formalization, realized within the COQ proof assistant, of the *Java Card* platform. This programming environment derived from *Java* is intended to smart cards and to their security requirements. More precisely, we describe the formalization of the *Java Card* virtual machine and of a substantial part of its runtime environment. Then, we explain how to obtain, from this virtual machine dynamically verifying type safety, a bytecode verifier (BCV) and a virtual machine more effective for execution but as safe. We bring for the BCV a generic framework, expressed with modules, applying the most recent techniques from this domain and simplifying proofs needed to insure soundness of the built BCV. Finally, we show how the methodology applied for type safety can be generalized and describe tools realized to automate this task.