Outils et techniques pour la vérification formelle de la plate-forme JavaCard

Simão Melo de Sousa

Thèse de doctorat

Février 2003

Remerciements

Lorsque je me retourne sur les trois ans que j'ai dédiés à ma thèse, je ne cesse de penser que, pour beaucoup, cette thèse a été l'œuvre d'un concours de circonstances. Je tiens donc à remercier chaleureusement tous ceux qui ont fait que celles-ci me soient favorables.

Tout d'abord, je voudrai remercier Gilles Barthe d'avoir accepté de diriger mes travaux de doctorat. Sa perspicacité et ses qualités de chercheur resteront un modèle pour moi.

J'aimerai ensuite remercier Laurence Pierre qui m'a fait l'honneur de présider mon jury. Je remercie également chaleureusement Christine Paulin-Mohring et José Nuno de Oliveira qui ont tout deux accepté d'être les rapporteurs de ma thèse. La pertinence de leur analyse et de leurs remarques fut précieuse. Je tiens aussi à remercier Jean-Louis Lanet et Xavier Leroy de l'intérêt qu'ils ont porté à mes travaux en acceptant d'être membre de mon jury.

Mes travaux de doctorat ont été conduits au sein de l'équipe Lemme. Je tiens donc à remercier toute l'équipe pour l'agréable ambiance, si importante à la conduite d'une telle tâche de longue haleine.

J'ai été co-financé par la FCT — *Fundação Para a Ciência e a Tecnologia*[1]- Portugal et par l'Université de la Beira Interior - Portugal. Ce point est essentiel à la bonne tenue d'une thèse. Qu'ils en soient remerciés.

La question du nom d'une de mes contributions a donné lieu à de longues discussions. J'ai toujours aperçu dans cette appellation l'enrichissant défi — et en cela je suis reconnaissant, de défendre le nom de JaKarTa au sein d'une communauté, à laquelle j'appartiens, plutôt historiquement sceptique envers les intérêts indonésiens. ;-)

Dans la partie finale de ma thèse, j'ai pu faire appel, avec bonheur, à un compilateur optimisé "françugais-français". Je tiens à en rendre grâce à ses concepteurs, M. et Mme Dufay. Oui, Guillaume, mais aussi Pierre, ont été d'infatigables et indéfectibles lecteurs et correcteurs de mon manuscrit. Travailler, partager un bureau avec eux a été déterminant pour moi et, au delà de cela, un réel plaisir. Pour tout cela, un très très grand merci !

Enfin, ces quelques années auraient été difficiles sans le concours de tous mes amis. Ils savent à quel point je leur attache de l'importance. Antonia, Carine, Fabrice, Franck, Frédéric, Guillaume, Henrik N., Henrik P., Herman S., mes amis de l'I3S, Julien, Ludovic, mes amis du Minho, Néstor, Pierre, Romain et

[1]Référence SFRH/BD/790/2000

mes amis de l'UBI, sachez que cette thèse vous doit beaucoup. Merci, merci à vous tous !

J'aimerai remercier mes proches sans qui, cela va sans dire, je n'en serais pas là et qui m'ont offert leur soutien indéfectible. Merci Maman et Papa. Merci Carla, Cecília, Cesar, Nelson et Aurora.

Pour finir, Manuela et petite Patrícia, merci pour tout, et plus encore.

Si j'avais su, au lieu de faire la guerre, j'aurais appris l'informatique

Xanana Gusmão, leader charismatique de la résistance timoraise au pouvoir de Djakarta, parlant de son élection à la présidence de Timor - Le Monde du 11/04/02

Table des matières

1 Synopsis **13**
1.1 Un survol de CertiCartes 17
1.2 Construction d'abstractions certifiées avec JaKarTa 21
1.2.1 Un survol de l'outil JaKarTa 22
1.2.2 Le Kit de Transformation JaKarTa 23
1.2.3 Synthèse de la machine virtuelle offensive 24
1.2.4 Synthèse de la machine virtuelle abstraite 24
1.2.5 Automatisation de la validation croisée des machines virtuelles 25
1.3 Plan de la thèse 25
1.3.1 Guide de lecture 26
1.4 Contributions 26

2 Les Cartes à Puce et JavaCard **29**
2.1 Le cadre 29
2.2 L'utilisation des cartes à puce 30
2.3 Qu'est ce qu'une carte à puce ? 33
2.3.1 Les différents standards 36
2.3.2 Les systèmes d'exploitation 37
2.4 La plate-forme JavaCard 38
2.5 Les cartes à puce sont-elles le Saint Gräal de la sûreté ? 42
2.5.1 La sécurité comme un processus complexe 43
2.5.2 La vérification formelle de la plate-forme JavaCard est elle justifiée ? 47

3 Les Méthodes Formelles **49**
3.1 Cadre et motivation 49
3.2 Les Critères Communs 51
3.3 Les Méthodes Formelles : Techniques et Outils 54
3.3.1 Spécifier et analyser 54
3.3.2 Spécifier et démontrer 57
3.3.3 Spécifier et dériver 60
3.3.4 Spécifier et transformer 62
3.3.5 Combiner le meilleur des mondes 62
3.3.6 Un rapide survol du système Coq 63
3.4 Modélisation et vérification formelle de Java et de JavaCard . . . 66

3.4.1 Vérification de la plate-forme Java/JavaCard 66
3.4.2 Vérification des applications Java/JavaCard 70
3.5 Conclusion 72

4 CertiCartes 75
4.1 Considérations préliminaires 76
4.2 Représentation des programmes JavaCard 77
4.2.1 Structures de données et valeurs 77
4.2.2 Les valeurs dynamiques 78
4.2.3 Représentation des programmes 80
4.3 Le JCVM Tools 83
4.3.1 Correction de la représentation 85
4.4 La Machine Virtuelle Défensive 85
4.4.1 Le modèle mémoire 86
4.4.2 Les Instructions 88
4.4.3 La gestion des Exceptions 89
4.4.4 Sémantique des instructions 92
4.5 Exécutabilité 101
4.5.1 Transformation des Fichiers 102
4.5.2 Un exemple 102
4.6 Une méthodologie pour l'abstraction de la machine virtuelle . . . 104
4.7 La machine virtuelle offensive 106
4.7.1 Équivalence des machines offensive et défensive 112
4.8 La machine virtuelle abstraite 115
4.8.1 Équivalence des machines abstraite et défensive 121
4.9 De la machine virtuelle abstraite au vérifieur de bytecode 124
4.10 Conclusion 124

5 JaKarTa 127
5.1 JSL : support à la spécification 129
5.1.1 La syntaxe abstraite JSL 130
5.1.2 Spécification JSL 134
5.1.3 Vérification du typage 136
5.1.4 L'outil JaKarTa 141
5.2 JTK : support à l'abstraction 144
5.2.1 Une présentation par l'exemple 144
5.2.2 Le processus de traduction syntaxique 147
5.2.3 Le processus de nettoyage 153
5.2.4 Le processus de transformation 153
5.2.5 Les commandes d'abstraction 156
5.2.6 Interprétation des commandes 158
5.3 JPI : support à l'interaction 160
5.3.1 Le format JIR 160
5.4 JAK : support à la preuve 162
5.4.1 Preuves par induction sur les fonctions définies par filtrage 162
5.4.2 Validation d'abstractions 164
5.5 Conclusion 165

5.5.1 Évaluation 165

6 Applications de JaKarTa à CertiCartes 167
6.1 La machine virtuelle défensive 168
6.1.1 La modélisation des données de la dJCVM 168
6.1.2 Un exemple d'instruction en syntaxe JSL 168
6.2 Synthèse de la machine virtuelle offensive 170
6.2.1 Modélisation des données de la oJCVM 171
6.2.2 Fonctions d'abstraction pour la oJCVM 171
6.2.3 Script d'abstraction pour la oJCVM 171
6.2.4 Résultat de la synthèse de la oJCVM 174
6.3 Synthèse de la machine virtuelle abstraite 175
6.3.1 Modélisation des données de la aJCVM 175
6.3.2 Fonctions d'abstraction pour la aJCVM 176
6.3.3 Script d'abstraction pour la aJCVM 177
6.3.4 Résultat de la synthèse de la aJCVM 180
6.4 Automatisation de la validation croisée des machines virtuelles . 182
6.5 Conclusion 183

7 Conclusion et Perspectives 185
7.1 CertiCartes 185
7.2 JaKarTa 187
7.3 Perspectives 189
7.3.1 CertiCartes 189
7.3.2 JaKarTa 190

A Indice des Notations 205

Résumé

La vérification du bytecode est l'une des pièces maîtresses de la sécurité de l'architecture JavaCard. Sa correction est souvent formulée en utilisant les notions de machine virtuelle défensive et de machine virtuelle offensive. La machine virtuelle défensive vérifie la cohérence des données manipulées pendant l'exécution alors que la machine offensive ne fait pas de telles vérifications. La correction est ainsi établie en vérifiant que l'exécution de programmes qui ont passé avec succès la vérification de bytecode, sur les deux machines, coïncide. Dans cette thèse nous décrirons comment sont spécifiées de telles machines virtuelles et comment sont définis de tels énoncés de correction dans les assistants de preuve. Nous nous concentrerons en particulier sur le problème de l'automatisation de la construction de machines virtuelles offensives et de vérifieurs de bytecode à partir de machines virtuelles défensives.

Abstract

Bytecode verification is one of the key security functions of the JavaCard architecture. Its correctness is often cast relatively to a defensive virtual machine that performs checks at run-time, and an offensive one that does not, and can be summarized as stating that the two machines coincide on programs that pass bytecode verification. We review the process of establishing such a correctness statement in a proof assistant, and focus in particular on the problem of automating the construction of an offensive virtual machine and a bytecode verifier from a defensive machine.

Chapitre 1

Synopsis

Les cartes à puce [133] sont de petits dispositifs créés pour contenir et manipuler des données confidentielles. Elles peuvent ainsi être utilisées comme cartes d'identification ou cartes d'accès à des applications ou services. Actuellement, elles sont de plus en plus utilisées par des organismes gouvernementaux ou commerciaux comme cartes bancaires, portefeuilles électroniques, cartes pour téléphones mobiles (appelées cartes SIM), dispositifs d'identification électronique (e-IDs), *etc*, et sont appelées à jouer un rôle clé dans le renforcement de la confiance et de la confidentialité de la monétique (e-payements) et de l'administration publique électronique (e-gouvernement) [62].

Les cartes à puce *Open Platform* constituent la nouvelle génération de cartes à puce qui fourni une plus grande flexibilité. De telles cartes à puce

- intègrent une machine virtuelle qui permet de s'abstraire des spécificités du système d'exploitation et de la couche matérielle. Ceci permet la programmation d'applications cartes à puce, ou *applets*, dans des langages de haut niveau. Les applets étant postérieurement compilées et chargées sur la carte à puce.
- sont multi-applicatives, dans le sens où elles permettent la coexistence sur le même support de plusieurs applications, fournissent des mécanismes de communication inter-applications sécurisés et, finalement, supportent le chargement d'applications après distribution.

La flexibilité des cartes à puce *Open Platform* est à la fois leur atout majeur et le plus grand obstacle à leur large déploiement. D'un coté la conception d'applets dans des langages de haut niveau réduit le coût et le temps de production, et la possibilité d'exécuter plusieurs applets sur une seule et même carte ouvre la voie à de nouveaux types d'application. De l'autre, de telles fonctionnalités introduisent la possibilité de charger des applets malicieuses qui pourront exploiter les faiblesses de la plate-forme, lire, modifier, effacer des données confidentielles d'autres applets ou encore de simplement bloquer la carte.

JavaCard et la Plate-Forme JavaCard. JavaCard [91], et en particulier la version 2.1 que nous abordons dans cette thèse, est le langage de programmation standard des cartes à puce multi-applicatives. Créé en 1996 par Sun, JavaCard apporte aux environnements à ressources limitées propres aux plates-formes cartes à puce les avantages de Java : l'orienté objet, la sûreté par typage, la portabilité. Voici une brève description de JavaCard et de ses différences avec Java :

- les applets JavaCard sont écrites dans un sous-ensemble du langage Java qui omet certaines caractéristiques comme les types larges, tableaux de tableaux, finalisation ou encore les processus légers (les threads). De plus, les applets JavaCard ne peuvent qu'utiliser les API JavaCard qui définissent et contraignent l'interface entre une applet et son environnement d'exécution sur la carte ;
- les applets JavaCard sont compilées dans le format *CLASS* et sont ensuite converties dans le format *CAP*. La conversion implique un certain nombre d'optimisations guidées par les contraintes propres aux cartes à puce. Par exemple, les noms sont remplacés par des tokens et les fichiers *.class* sont fusionnés en paquets appelés *packages* ;
- les applets JavaCard passent, après compilation, par un vérifieur de bytecode (BCV) qui vérifie d'une part si les applets sont correctement formées et bien typées, et d'autre part si l'exécution de ses dernières n'inclue pas l'exécution d'opérations malicieuses [106]. Jusqu'à assez récemment, la vérification de bytecode était faite avant que l'application soit chargée sur la carte. En cas de succès de la vérification, l'applet était en mesure d'être signée et chargée sur la carte. Cependant, les avancées récentes sur la phase de vérification du bytecode permettent maintenant de placer le vérifieur à même la carte, et ainsi de recentrer l'architecture de sécurité sur la carte elle-même. Il semble qu'une telle architecture jouera un rôle déterminant dans le domaine des cartes à puce [42, 40, 107] ;
- les applets JavaCard sont exécutées par l'environnement d'exécution JavaCard (JCRE). Le JCRE comprend la machine virtuelle JavaCard (JCVM), fournit le support aux APIs JavaCard et invoque les services de la couche native lorsqu'est nécessaire l'exécution de tâches de bas niveau comme la gestion des communications. La JCVM est une machine abstraite à pile comme la machine virtuelle Java (JVM), cependant certaines fonctionnalités de la JVM comme le chargement dynamique de classes ne sont pas supportées ;
- JavaCard introduit des fonctionnalités qui ne sont pas présentes, du moins pas sous la même forme, dans Java. Par exemple, JavaCard 2.1 introduit les mécanismes de pare-feu permettant l'isolation des applets, et JavaCard 2.2 introduit les mécanismes des canaux logiques et un forme simplifiée des appels de méthodes distantes (*remote method invokation*, RMI).

Une présentation plus détaillée et technique de JavaCard peut être trouvée dans [43, 153, 150, 151, 152].

Méthodes formelles pour les cartes à puce. Il est communément admis qu'un haut niveau de fiabilité est requis pour qu'une large acceptation des cartes à puce comme dispositif informatique sûr soit possible. Il est aussi admis que l'utilisation des méthodes formelles dans la conception, le test et la vérification des plates-formes pour cartes à puce et de ces applications permettent d'atteindre les plus hauts de ces niveaux de fiabilité. Ce point de vue est aussi celui présenté par les Critères Communs (plus connus sous leur dénomination anglaise *Common Criteria*) [55], qui définissent un schéma unifié pour l'évaluation de la sécurité des produits issus des technologies de l'information (IT). Les Critères Communs imposent, par exemple, que l'évaluation pour les plus haut niveaux (les niveaux EAL 5 à EAL 7) soit fortement basée sur les méthodes formelles.

Il n'y a pas une mais plusieurs vérifications formelles pour cartes à puce. Chacune d'elles a ses propres objectifs et ses propres techniques.

Par exemple :

- la vérification de programmes au niveau des sources permet d'aider les développeurs à assurer que leurs applets sont correctes. Le point fort d'une telle approche est que le développeur, ou un expert en méthodes formelles participant au développement, peut fournir une spécification formelle du programme et vérifier/prouver que l'applet est correcte par rapport à sa spécification. Il existe une grande diversité de formalismes permettant l'expression de telles spécifications. Les logiques des prédicats, temporelles ou dynamiques, ou encore les langages de spécification comme JML [93] sont des exemples de tels formalismes et les techniques de vérification associées incluent la vérification pendant la phase d'exécution, le test d'unité [44], les tests statiques [104], la vérification de modèles [51], les démonstrateurs de théorèmes interactifs [26] ;
- la vérification des programmes compilés, c'est-à-dire du bytecode, peut aider à vérifier s'il est sûr de charger et d'exécuter une applet dont l'origine est inconnue ou douteuse. Comme il n'est pas raisonnable de déléguer au consommateur la tâche de vérifier les applets chargées sur sa carte, il est important que ce type de vérification utilise des techniques automatiques comme les analyses statiques, la vérification de modèles [31, 129], ou s'appuie sur un schéma dans lequel la phase de vérification est déléguée au producteur de code [122].

La vérification de la Plate-Forme, qui est au cœur de cette thèse, tente d'établir qu'une plate-forme pour cartes à puce est correcte en ce sens qu'elle adhère à une politique de sécurité acceptable. Une telle politique est usuellement formulée en terme d'ensemble de contraintes comme par exemple "une référence ne peut être convertie en un entier", ou "la machine virtuelle ne peut sauter vers une adresse arbitraire". Dans le cas de la plate-forme JavaCard, un tel ensemble de contraintes devra inclure des contraintes relatives à la sûreté du typage et à l'isolation des applets. L'objectif de la vérification de plate-forme est de prouver formellement que ces contraintes sont garanties par des fonctions de sécurité bien définies. Une de ces importantes fonctions de sécurité est le vérifieur de bytecode, qui garantit des propriétés comme la correction des types, la contention du code, la correcte initialisation des objets ou encore l'absence de débordement de pile. La preuve de la correction du vérifieur de bytecode est un pré-requis pour les plus hautes évaluations Critères Communs et constitue la ligne directrice de nos travaux.

La correction de la vérification du bytecode est énoncée comme une correspondance entre différents types de machines virtuelles [41, 102, 135] :

- entre une machine virtuelle JavaCard défensive (JCVM défensive), qui manipule des valeurs typées et qui exécute des vérifications de type et structurelles pendant l'exécution. De telles machines défensives sont proches des spécifications de SUN mais inefficace en terme de vitesse d'exécution et de consommation de mémoire ;
- et une machine virtuelle JavaCard offensive (JCVM offensive), qui manipule des valeurs non typées et qui repose sur le succès de la phase de vérification de bytecode pour pouvoir éliminer, pendant l'exécution, les tests de vérification propres à la machine défensive. En ce sens les machines virtuelles offensives sont proches des implémentations existantes.

L'énoncé de correction peut être résumé par la phrase suivante : "Les machines virtuelles JavaCard offensive et défensive coïncident sur les programmes qui ont passés avec succès la phase de vérification de bytecode".

Formuler précisément et prouver un tel énoncé nécessite :

- la modélisation des machines virtuelles défensive et offensives, et la formalisation de la notion de correspondance entre les deux machines par le biais d'une fonction d'abstraction α_{do} qui transforme les états de la machine défensive en états de la machine offensive.
- la preuve que les deux machines coïncident sur les programmes dont l'exécution dans la machine défensive n'a pas levé d'erreur de type ;
- la modélisation du BCV comme une analyse du flot de données d'une machine abstraite virtuelle qui ne manipule que des types, et la preuve que la machine défensive ne lance pas d'erreur de type pour des programmes qui ont passé avec succès la phase BCV.

Dans le contexte des évaluations Critères Communs de haut niveau, il est requis que de tels modèles et preuves soient nécessairement formels, et, de fait, ceux-ci sont mieux mis en œuvre avec l'aide d'assistants de preuve.

CertiCartes [21, 22] est une mise en œuvre complète de la preuve de correction de la verification de bytecode pour JavaCard 2.1, réalisée avec l'assistant de preuve Coq [50]. Actuellement CertiCartes comprend une spécification des trois machines virtuelles JavaCard (c'est-à-dire défensive, offensive, abstraite) et une spécification du BCV. Le développement Coq est structuré en deux modules séparés :

1. le premier module inclut la construction des trois machines virtuelles ainsi que leur validation croisée[1]. Ce développement représente la clef de voûte de l'architecture de CertiCartes ;
2. le second module, le BCV, comprend la construction et la validation du vérifieur de bytecode. Ce développement est largement indépendant des définitions des machines virtuelles, qui sont considérées comme des paramètres dans la construction du BCV.

Une évaluation externe effectuée par les laboratoires de recherche Gemplus [101] suggère que CertiCartes modélise la plate-forme JavaCard à un niveau

[1]La validation de chaque machine virtuelle est croisée en ceci qu'elle dépend de celle des autres.

raisonnable de détail, et propose un certain nombre modifications concrètes pour permettre l'évaluation Critères Communs de CertiCartes. Ce n'est pas sur ce point précis que nous avons décidé de poursuivre notre effort, en particulier parce que la tâche est considérable. Giménez et Ly [74] rapporte qu'un travail de six hommes/an a été nécessaire au développement d'une formalisation de JavaCard 2.1 contemplant un niveau satisfaisant de détails.

Évaluation Notre expérience avec CertiCartes corrobore le point de vue partagé par la communauté des méthodes formelles que la vérification assistée par ordinateur des langages de programmation standard est devenue viable mais néanmoins souffre d'un manque d'outils appropriés aidant la spécification et la preuve de grande taille. Un des aspects les plus intéressants de CertiCartes est la validation d'une méthodologie originale qui est bien adaptée au raisonnement sur des langages de bas niveau, et qui peut être appliquée aux systèmes de types pour Java(Card) [13, 67, 103, 147].

De ce fait, nos travaux suggèrent qu'il est possible de développer des outils appropriés à cette méthodologie et, en particulier, appropriés à la construction de machines virtuelles offensives et abstraites à partir de machines défensives ainsi qu'à l'établissement de leur validation croisée.

JaKarTa [20] est une boite à outils expérimentale pour la spécification et le raisonnement sur les plates-formes JavaCard. L'objectif principal est de fournir :

- une interface, indépendante de celles des différents outils de preuve, permettant la description de machines virtuelles et fournissant des outils de transformation de telles descriptions comme les abstractions, les raffinements ou encore les optimisations ;
- un compilateur traduisant ces spécifications dans un format utilisable par les assistants de preuve, ainsi que des outils facilitant la preuve de correction des transformations dans ces assistants.

Notre attention s'est surtout portée sur les outils permettant la construction (1) de machines virtuelles JavaCard offensives et abstraites à partir de machines défensives ; (2) et de leur respective validation. On peut penser que ce but est trop spécifique, mais la machine défensive a de très nombreuses instanciations qui sont déterminées par la propriété qui doit être assurée statiquement par le typage. Ces propriétés ont, typiquement, pour sujet l'initialisation des objets, le flux d'information, le contrôle des ressources, les différents blocages, *etc.* Dans cette thèse nous aborderons la sûreté du typage comme cas d'étude de ces propriétés.

1.1 Un survol de CertiCartes

Comme nous l'avons déjà indiqué, CertiCartes est structuré en deux modules. Le premier module, que nous référerons par JCVM, comprend la construction de trois machines virtuelles et leurs validation croisée. Le deuxième module, nommé BCV, comprend la construction et la validation d'un vérificateur de bytecode construit au dessus de la machine abstraite. Nous allons maintenant décrire brièvement le contenu de CertiCartes. De plus amples détails sont fournis dans le chapitre 4 et dans [17, 21, 22].

Module JCVM

Modélisation des programmes. Pour des raisons de portabilité, les programmes sont formalisés en utilisant un style mathématique délibérément neutre, basé sur les types enregistrement, les types inductifs (non-dépendants et de premier ordre) et basé sur les mécanismes associés comme l'analyse par cas, la récursion structurelle. Par exemple les programmes bytecode JavaCard sont représentés par l'enregistrement suivant :

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

où les types `Interface`, `Class` et `Method` sont eux même définis par des types enregistrements. Le champ `static_heap_type` regroupe les types des champs statiques déclarés au sein du programme considéré. Pour des raisons de simplicité, nous préférons manipuler des programmes clos. Dès lors les parties nécessaires des packages `java.lang` et `javacard.framework` font partie intégrante des programmes.

Modélisation de la mémoire. La mémoire est modélisée de façon semblable. Par exemple, les états d'une machine virtuelle sont formalisés par un enregistrement collectant le tas (qui contient les objets créés pendant l'exécution), l'image des champs statiques (qui contient les valeurs des champs statiques des classes) et la pile de frames (pile d'exécution où les frames sont des environnements d'exécution de méthodes). Formellement :

```
Record dstate := {
  hp : heap;
  sh : sheap;
  sk : stack;
}.
```

Pour pouvoir prendre en compte la terminaison brutale de l'exécution (qui peut survenir lorsqu'il n'est pas possible de récupérer une exception lancée ou lorsqu'un programme s'avère mal formé), nous introduisons le type des états de retour, définis par un type somme :

```
Inductive rdstate :=
  Normal   :              dstate → rdstate |
  Abnormal : exception → dstate → rdstate.
```

La définition des états (de retour) et la définition de toutes les composantes du modèle de la mémoire sont paramétrées par le type générique des valeurs que nous laissons non spécifié. Le modèle mémoire sera instancié plus tard vers les machines défensive, offensive et abstraite. Le tableau suivant introduit la notion de valeur associée à chaque type de machine virtuelle[2].

Machine Virtuelle	Défensive	Offensive	Abstraite
Valeur	(Type,Nombre)	Nombre	Type

Modélisation de la machine virtuelle défensive. Premièrement nous instancions le modèle de la mémoire avec la notion de valeur typée. Nous obtenons ainsi le type des états des JCVMs défensives `dstate` ainsi que les états de retour des JCVMs défensives `rdstate`.

Nous modélisons, ensuite, la sémantique défensive de chaque bytecode JavaCard `b` comme un fonction `dexec_b: dstate` → `rdstate`. Classiquement, la fonction `dexec_b` extrait de l'état les valeurs dont le bytecode a besoin, vérifie le type de ces valeurs et met à jour l'état avec les résultats de l'exécution du bytecode.

Finalement, la fonction calculant un pas d'exécution de la machine défensive est défini par `dexec: dstate` → `rdstate`. Cette fonction extrait de l'état considéré le bytecode JavaCard `b` qui doit être exécuté et appelle la fonction `dexec_b` correspondante.

Modélisation de la machine virtuelle offensive. Là aussi nous commençons par instancier le modèle de mémoire avec la notion adéquate de valeur. Dans le cas de la machine virtuelle offensive il s'agit des valeurs non-typées. On obtient ainsi le type des états offensifs `ostate` (respectivement des états de retour offensifs `rostate`). À l'image de `dexec`, la fonction calculant un pas d'exécution est modélisée par `oexec: ostate` → `rostate`, mais, dans ce cas ci, les vérifications de type sont omises.

Modélisation de la machine virtuelle abstraite. La machine virtuelle abstraite qui est utilisée par le BCV ne manipule que des types. Le modèle de mémoire est dès lors simplifié car :

1. la machine abstraite opère méthode par méthode, ce qui évite d'avoir à considérer l'empilement des contextes d'exécution (l'exécution dans ce cadre n'a besoin que d'un seul contexte d'exécution) ;
2. le type des objets est stocké dans la pile d'opérande ce qui évite d'avoir à considérer le tas d'objets ;
3. L'image des champs statiques devient une liste de types. Or le champ `static_heap_type` de la représentation des programmes (de type `jcprogram`) détient déjà cette information. Nous n'avons donc pas besoin de considérer le tas statique.

Formellement nous définissons donc le type `astate` des états abstraits comme celui des contextes d'exécution abstraits.

La fonction calculant un pas d'exécution est, dans sa version abstraite, non-déterministe car les instructions de branchement conduisent à différents points

[2]Dans la section 4.2.1 nous reviendrons sur l'exact encodage choisi pour les valeurs typées.

de programme et dès lors produisent non plus un seul état mais plusieurs. Nous utilisons une liste regroupant tous les états de retour possibles. Dès lors la fonction d'exécution est modélisée par `aexec: astate` → `(list rastate)`, où `astate` est le type des états de retour.

La validation croisée des machines virtuelles JavaCard est un prérequis à l'établissement de la correction du BCV et consiste à montrer (i) que les machines virtuelles offensive et défensive coïncident pour des programmes qui ne lancent pas d'erreur de type lorsqu'ils sont exécutés sur la machine défensive (ii) que tout programme qui lance une erreur de type lorsqu'il est exécuté par la machine virtuelle défensive lance également une erreur de type dans la machine virtuelle abstraite.

L'interprétation abstraite offre un cadre élégant à l'expression de telles propriétés. Ainsi les machines virtuelles offensive et abstraite peuvent être vues comme des abstractions de la machine virtuelle défensive. Dès lors les énoncés de correction peuvent être également considérés comme l'expression de la cohérence des abstractions menant à ces machines. Formellement nous définissons deux fonctions d'abstraction :

```
alpha_do : dstate → ostate
alpha_da : dstate → astate
```

conjointement avec leur homologue sur les types des états de retour :

```
alpha_do_rs : rdstate → rostate
alpha_da_rs : rdstate → rastate
```

et nous prouvons, sous des contraintes adéquates, que les diagrammes des figures 1.1 et 1.2 commutent (où la flèche du coté droit de la figure 1.2 signifie que l'abstraction de l'état de retour défensif est, à sous-typage près, un élément de la liste des états de retour).

$$
\begin{tikzcd}
\texttt{dstate} \arrow[r, "\texttt{dexec}"] \arrow[d, "\texttt{alpha\_do}"] & \texttt{rdstate} \arrow[d, "\texttt{alpha\_do\_rs}"] \\
\texttt{ostate} \arrow[r, "\texttt{oexec}"] & \texttt{rostate}
\end{tikzcd}
$$

FIG. 1.1 – DIAGRAMME DE COMMUTATION DES EXÉCUTIONS DÉFENSIVE ET OFFENSIVE

La commutation des deux diagrammes est prouvée par une analyse par cas du bytecode à être exécuté puis par une analyse des états possibles de la machine virtuelle défensive et finalement par raisonnement équationnel.

En résumé, le module JCVM représente la plupart de l'effort fourni pour CertiCartes. Toutefois, la construction des machines offensive et abstraite reposent sur une interprétation abstraite et peut être automatisée. De la même façon, la validation croisée des machines virtuelles se fondant sur des raisonnements par cas et équationnels, peut aussi être automatisée.

$$
\begin{tikzcd}
\texttt{dstate} \arrow[r, "\texttt{dexec}"] \arrow[d, "\texttt{alpha\_da}"] & \texttt{rdstate} \arrow[d, hook, "\preceq\texttt{alpha\_da\_rs}"] \\
\texttt{astate} \arrow[r, "\texttt{aexec}"] & (\texttt{list rastate})
\end{tikzcd}
$$

Fig. 1.2 – Diagramme de commutation des exécutions défensive et abstraite

Module BCV

Développé par G. Dufay [22, 59, 17], ce module construit un vérificateur de bytecode qui repose sur la machine virtuelle abstraite en utilisant une formalisation de l'algorithme de Kildall, similaire à celle présentée dans [97, 124].

Cette formalisation prend en entrée :

- un type `astate` des états abstraits et une relation bien fondée `lt_astate` sur `astate` ;
- une machine virtuelle abstraite `aexec: astate` → `(list astate)` qui est monotone par rapport à `lt_astate` ;

et produit en sortie :

- une fonction `bcv: jcprogram` → `bool` ;
- une preuve qu'un programme `p` tel que `bcv p = true` ne lance aucune erreur de type lorsqu'il est exécuté par la machine virtuelle abstraite `aexec`.

De plus, ce module contient une justification de la méthodologie de vérification "méthode par méthode", c'est-à-dire une preuve qu'un programme est bien typé si les méthodes le constituant, individuellement, le sont. Conjointement avec la preuve de la commutation des diagrammes, ces résultats garantissent la correction de la vérification du bytecode.

En guise de conclusion, remarquons que le module BCV est largement indépendant des considérations qui ont mené la conception des machines virtuelles considérées. Dès lors la méthodologie sous-jacente au module BCV peut être appliquée à d'autres cadres sans trop de problèmes.

1.2 Construction d'abstractions certifiées avec JaKarTa

L'objectif de JaKarTa est de fournir un environnement interactif pour la transformation de machines virtuelles et la vérification de la correction de cellesci dans des systèmes de preuve. Des exemples de telles transformations sont les raffinements, les optimisations, les abstractions, mais notre contribution ne traitera que le support aux abstractions.

Dans ce qui suit nous présentons les mécanismes d'abstraction de JaKarTa et décrivons comment ils ont été utilisés pour dériver les machines virtuelles JavaCard offensive et abstraite à partir de la machine virtuelle défensive.

Objectifs

Outre (et en conséquence de) la construction de CertiCartes, les objectifs poursuivis dans cette thèse sont essentiellement de fournir un support automatique à la construction de machines virtuelles pour une propriété arbitraire P, et de mettre en place l'encadrement nécessaire à la validation croisée de ces machines virtuelles. Concrètement nous voulons que l'outil JaKarTa puisse, à partir des données suivantes :

- une machine virtuelle P-défensive `dexec: dstate` → `dstate` qui manipule des valeurs typées (la notion de type étant déterminée par P) et qui garantit P par une vérification dynamique (au cours de l'exécution) ;
- deux fonctions d'abstraction :

  ```
  alpha_da : dstate → astate
  alpha_do : dstate → ostate
  ```

  où `astate` et `ostate` dénotent les types des états abstraits et offensifs adéquats ;
- un script de transformation, c'est-à-dire une séquence de commandes indiquant comment conduire la dérivation de la machine P-défensive vers les machines P-offensive et P-abstraite ;

produire une machine P-offensive et une machine P-abstraite :

```
oexec : ostate → orstate
aexec : astate → (list arstate)
```

couplées aux preuves qu'elles résultent de transformations cohérentes de la machine P-défensive.

1.2.1 Un survol de l'outil JaKarTa

Le Langage de Spécification JaKarTa, JSL, est un langage simple permettant la description de machines virtuelles dans un style mathématique neutre. JSL est un langage typé (polymorphe) qui possède un modèle d'exécution basé sur la réécriture [9, 30].

Une théorie JSL est donnée par un ensemble de déclarations qui introduisent des types de données polymorphes de premier ordre avec leurs constructeurs, et par un ensemble de définitions de fonction qui introduisent les symboles de fonction et fixent leur sémantique opérationnelle via un ensemble de règles de réécriture conditionnelles de la forme :

$$\mathtt{l}_1\rightarrow\mathtt{r}_1,\ldots,\mathtt{l}_n\rightarrow\mathtt{r}_n\Rightarrow \mathtt{g} \rightarrow \mathtt{d}$$

Le format JSL des règles de réécriture acceptables impose que les membres droits des conditions, c'est-à-dire les $\mathtt{r}_i$, soient des motifs ne contenant que des variables fraîches (rappelons qu'un motif est un terme linéaire construit à partir de variables et de symboles de constructeur). Un tel format peut paraître inhabituel mais permet de rester proche des formalismes avec filtrage propre aux systèmes de preuve et à la programmation fonctionnelle.

L'Interface de Preuve JaKarTa, JPI, permet la traduction des spécifications JSL vers les langages des systèmes de preuve ou des démonstrateurs automatiques. Actuellement les traductions vers Coq, Isabelle [126], PVS [142] et Spike [34] sont supportées. Les mécanismes de traduction peuvent prendre en compte des programmes qui seront considérés mal formés par le système cible (certains assistants de preuve n'acceptent pas les fonctions non-déterministes ainsi que les fonctions partielles), mais cela ne s'est pas révélé problématique en pratique.

Par ailleurs, JaKarTa permet la traduction de Coq vers JSL dans le but de pouvoir réutiliser les développements Coq existants (modulo certaines conditions) et vers OCaml [108] afin de permettre une exécution efficace des spécifications JSL.

Le Kit de Transformation JaKarTa, JTK, contient le moteur de dérivation d'abstraction de spécifications JSL (voir section 1.2.2).

Le Kit d'Automatisation JaKarTa, JAK, a, finalement, pour objectif de faciliter le processus de validation croisée des machines virtuelles tel qu'il est mis en œuvre dans les assistants de preuve (voir section 1.2.5).

1.2.2 Le Kit de Transformation JaKarTa

Principes L'abstraction $\hat{\mathtt{S}}$ d'une théorie JSL `S` est construite en trois étapes successives :

1. l'utilisateur définit, pour chaque type de donnée `T` de `S` le type de donnée abstrait correspondant $\hat{\mathtt{T}}$ avec la fonction d'abstraction $\alpha_{\mathtt{T}}$ de `T` vers $\hat{\mathtt{T}}$; de telles fonctions sont écrites en JSL, mais le format des règles est restreint à la réécriture inconditionnelle ;
2. l'utilisateur écrit un script contenant des instructions guidant le processus d'abstraction là où les procédures automatiques ne sont pas les plus adéquates. Une description plus détaillée des fonctionnalités et des effets d'un tel script sera donnée plus loin dans cette section et au chapitre 5 ;
3. en utilisant les données fournies par les points antérieurs, le JTK construit automatiquement, pour chaque fonction $\mathtt{f:T}_1 \rightarrow \mathtt{...} \rightarrow \mathtt{T}_n \rightarrow \mathtt{U}$ de `S` sa contrepartie abstraite $\hat{\mathtt{f}}\mathtt{:}\hat{\mathtt{T}}_1 \rightarrow \mathtt{...} \rightarrow \hat{\mathtt{T}}_n \rightarrow \hat{\mathtt{U}}$. À moins d'une contre-indication présente dans le script d'abstraction, la fonction $\hat{\mathtt{f}}$ est construite par une traduction syntaxique des règles définissant `f`, suivie d'une phase de nettoyage effaçant les conditions et les règles qui ont perdu leur sens.

La plupart du temps, le processus d'abstraction peut être conduit automatiquement, mais il existe cependant des configurations dans lesquelles l'intervention de l'utilisateur est nécessaire. C'est le propos des scripts d'abstraction de matérialiser l'intervention requise, et ceci par l'utilisation d'un ensemble de commandes bien précises.

Les commandes se divisent en trois catégories :

les commandes de suppression spécifient quels motifs dans les expressions ne sont pas désirés et doivent être effacés. Ces commandes sont utiles, par exemple, pour compléter l'oubli des opérations et vérifications du typage dans le processus de dérivation de la machine offensive JavaCard ;

les commandes de substitution spécifient les coercions et les substitutions qui devront être réalisées pendant la phase de traduction des règles de réécriture. Si nous prenons l'exemple de la dérivation de la machine virtuelle abstraite, de telles commandes sont utiles pour indiquer le remplacement de la recherche dynamique de méthodes par une recherche statique adéquate à la sémantique abstraite des bytecodes d'invocation de méthodes ;

les commandes de transformation agissent sur les fonctions qui, par abstraction, ont acquis un comportement que l'on désirerait modifier. C'est le cas, par exemple, des fonctions non-déterministes ou partielles. Une transformation classique calculée par ces commandes dans le cas d'une fonction non déterministe est la collecte pour une entrée donnée des valeurs possibles de retour dans une liste.

1.2.3 Synthèse de la machine virtuelle offensive

Disposant d'une spécification JSL de la JCVM défensive (constituée environ de 10000 lignes de code) produite à partir de la spécification Coq, nous avons été en mesure, en suivant les trois étapes décrites dans la section 1.2.2 de générer une machine offensive (d'environ 5000 lignes).

Le script utilisé pour générer la JCVM offensive implémente fidèlement le procédé qui a conduit, dans le cadre de la spécification Coq de CertiCartes, à la machine offensive (se reporter aux sections 1.1 et 4.7).

Dans un premier temps, sont définies les fonctions implémentant la transformation (l'abstraction) des données défensives vers leur contrepartie offensive. Celles-ci consistent essentiellement et sans surprise à retirer toute information de type dans le modèle de mémoire. Cette information suffit au JTK pour qu'il soit en mesure de produire la plupart de la sémantique offensive des instructions.

Les ajustements nécessaires sont de deux ordres : (1) il faut à certains endroits explicitement indiquer que les appels de fonction s'y trouvant correspondent à des vérifications de types qui sont, de fait, devenues inutiles ; (2) certaines données de base voient leur type être altéré dans la machine offensive sans que pour autant certaines des opérations les manipulant ne soient modifiées par l'abstraction. Pour celles-ci il est nécessaire d'indiquer une opération de coercion résolvant les conflits de typage qui en découlent.

Ceux-ci sont spécifiés par une dizaine de commandes de suppression et de substitution. Aucune commande de transformation n'a été nécessaire pour cette abstraction.

1.2.4 Synthèse de la machine virtuelle abstraite

De façon similaire, une machine abstraite (dont la taille avoisine les 5000 lignes) a été extraite de la machine défensive.

Une partie des subtilités propres à la machine virtuelle abstraite se retrouve capturée dans la définition des fonctions d'abstraction. Ainsi :

- dans la représentation des valeurs, les valeurs numériques sont supprimées pour ne subsister que l'information du type ;
- il y a cependant une exception notable : la machine virtuelle abstraite doit garder la trace des adresses de retour nécessaire à la gestion des sous-routines ;
- la pile de contextes d'exécution fait place au contexte d'exécution placé à son sommet ;
- les valeurs abstraites ne comportent aucun composant correspondant au tas des objets.

Mais même si les fonctions d'abstraction suffisent à synthétiser une partie significative de la spécification abstraite, les transformations exigées, par la nature même de l'abstraction, sont plus subtiles et ont requis plus d'ajustements que dans le cas de la machine offensive.

En particulier :

- l'introduction du non-déterminisme par les instructions de branchements (puisque ces instructions décident de la prochaine instruction à exécuter à partir de l'analyse de certaines valeurs) nécessite du traitement dispensé par les commandes de transformation ;
- certains comportements, dit dynamiques (par le fait de manipuler des données calculées à partir des valeurs de la machine) ne sont plus reproductibles tels quels dans la machine abstraite. Il faut modifier ces comportements de façon à ce qu'ils puissent s'adapter aux informations statiques disponibles.

Les commandes de suppression et de substitution traitent avec succès des autres ajustements.

1.2.5 Automatisation de la validation croisée des machines virtuelles

Afin d'automatiser les preuves de validation croisées des trois machines virtuelles à l'aide de Coq, JaKarTa fournit

- un module de génération de lemmes de commutation. Celui-ci analyse le processus d'abstraction d'une fonction JSL, et infère automatiquement le lemme de commutation pour cette fonction ;
- un ensemble de tactiques qui génèrent des principes d'induction pour le raisonnement sur des fonctions récursives complexes. Ces tactiques apportent un réel gain de productivité à l'effort de preuve. En particulier les scripts de preuve sont significativement plus faciles à construire et leur taille est réduite de près de 90% [16].

1.3 Plan de la thèse

Nous commencerons au chapitre 2 par une présentation des cartes à puce et de la plate-forme JavaCard. Nous aborderons dans ce chapitre la question de la sécurité de ce dispositif ainsi que le rôle des méthodes formelles, plus particulièrement de la vérification formelle, sur ce point.

Au chapitre 3 nous présenterons plus en détail les méthodes formelles. Pour commencer, nous aborderons la norme “Critères Communs” qui a été à l'origine des travaux présentés dans cette thèse, puis nous dresserons un inventaire des différentes méthodes formelles. Enfin nous donnerons un état de l'art de la vérification formelle de la plate-forme JavaCard.

Le chapitre 4 expose CertiCartes, la sémantique formelle exécutable de JavaCard que nous avons développée. Nous y aborderons la construction et la validation croisée des trois machines virtuelles (dJCVM, oJCVM et aJCVM) ainsi que la mise en place de cette architecture dans le cycle de production d'une applet JavaCard. Nous exposerons succinctement le JCVM Tools, qui est un outil utilisé dans le cycle de traduction de programmes JavaCard en Coq.

Nous aborderons au chapitre 5 les fondements de l'outil JaKarTa. Nous y décrirons les différents modules le constituant. Nous aborderons plus particulièrement le support à la spécification et le support à l'abstraction fourni par JaKarTa.

Nous exposerons au chapitre 6 l'utilisation de JaKarTa dans le cadre de CertiCartes. Nous décrirons la version JSL de CertiCartes puis montrerons comment nous avons obtenu automatiquement avec JaKarTa les machines aJCVM et oJCVM. Nous exposerons aussi l'aide apportée par JAK aux preuves de correction que nous avons mis en place.

Nous donnerons finalement nos conclusions au chapitre 7.

1.3.1 Guide de lecture

Le présent chapitre présente un résumé de la thèse. Les deux chapitres suivant forment le contexte et l'état de l'art et peuvent être omis lors d'une première lecture. Les chapitres 4 et 5, qui introduisent chacun une contribution indépendante, peuvent être lus séparément. Ceux-ci sont cependant nécessaires à la compréhension du chapitre 6.

1.4 Contributions

Les travaux présentés dans cette thèse représentent un travail d'équipe auquel nous avons contribué. Nous soulignons dans cette section nos contributions par rapport aux contributions de chaque membre de l'équipe.

CertiCartes Nous avons développé la sémantique formelle exécutable de JavaCard en étroite collaboration avec G. Dufay. Cette formalisation a été reprise, optimisée puis a servi de base aux travaux de G. Dufay sur la construction d'un vérificateur de bytecode. Nos contributions à CertiCartes peuvent se résumer en l'effort de modélisation de la JCVM défensive. G. Dufay a, pour sa part, été l'architecte de la structure des preuves, auxquelles nous avons apporté notre aide dans une version préliminaire. Le JCVM Tools a été, quant à lui, développé par B. Serpette.

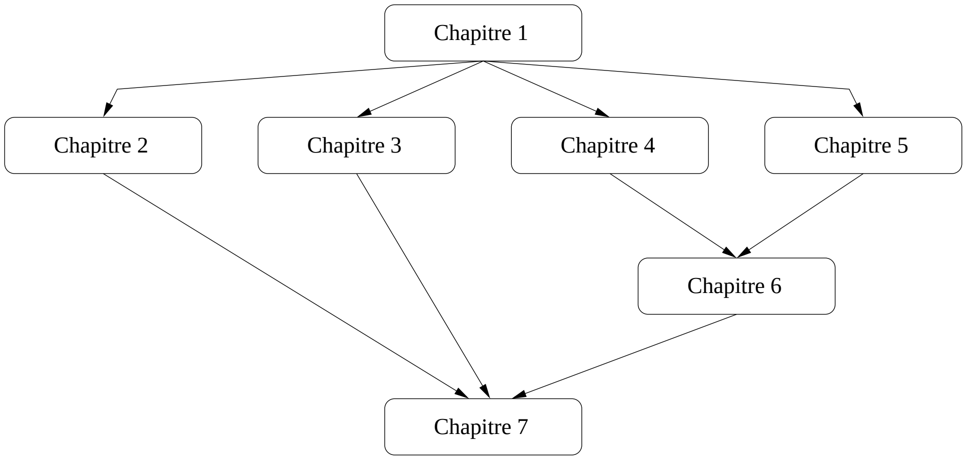

FIG. 1.3 – Dépendance des chapitres.

JaKarTa Nous avons été le principal concepteur de l'outil JaKarTa. Signalons cependant la contribution de P. Courtieu, concernant le support à la preuve fourni par le module JAK, que nous présenterons au chapitre 5.

Construction automatisée d'abstractions avec JaKarTa Le matériel exposé au chapitre 6 constitue une autre de nos contributions. Ajoutons que nous avons développé une première version des preuves de commutation devenue obsolète par l'apport du module JAK.

Chapitre 2

Les Cartes à Puce et JavaCard

Les cartes à microprocesseurs, communément appelées cartes à puces, sont de petits dispositifs présentés sous la forme d'un micro-module capable de contenir et manipuler des données. Cette thèse rapporte une étude sur la fiabilité et la sécurité d'une couche logicielle, la plate-forme JavaCard, placée sur de telles cartes.

L'objet de ce chapitre est, premièrement, de préciser ce que nous désignons par cartes à puce et par JavaCard, puis de placer ces notions dans leur contexte afin d'en décrire les enjeux en termes de sécurité. Une exposition complémentaire et approfondie des concepts abordés dans ce chapitre peut être trouvée dans [39, 43, 80, 81, 83, 111, 133].

2.1 Le cadre

L'explosion des technologies internet, de l'informatique omniprésente et de la communication digitale préfigurent l'avènement des sociétés digitales interconnectées.

L'homme mobile Les sociétés actuelles et les besoins qu'elles engendrent évoluent rapidement. Le citoyen d'aujourd'hui devient de plus en plus mobile mais désire avoir accès, où qu'il se trouve et au niveau de qualité auquel il est habitué, à tous les services auxquels il adhère. Naturellement, il est aussi exigent sur la qualité et la décentralisation des services offerts. Mieux encore, *l'homme mobile* aimerait voir sa mobilité stimulée et sécurisée au maximum.

L'homme digital De l'autre bout de la chaîne, les fournisseurs de services (comme les administrations publiques ou le secteur de la communication mobile etc...) doivent s'adapter aux nouvelles exigences et aux nouveaux marchés. Il devient essentiel de systématiser les traitements autant que possible (fournir un traitement — de masse, de qualité) mais aussi d'acquérir un contrôle plus fin et plus sûr des ressources impliquées. Nous nous acheminons ainsi vers un monde où à chaque client (un citoyen ou tout autre "entité") correspond son pendant numérique, c'est l'avènement de *l'homme digital*.

L'informatique omniprésente Les progrès technologiques de ces dernières années ont permis une miniaturisation, une dissémination du support informatique à tous les niveaux de la vie quotidienne. Ainsi si les nouvelles technologies apportent une réponse aux exigences de la modernité, elles formulent aussi des solutions inédites servant aussi de moteur à l'évolution. L'*informatique omniprésente* est l'un des paradigmes acteurs de la modernité. Force est de constater le bien être et la sécurité que peuvent apporter ce qui n'étaient, il n'y a pas si longtemps que cela, que des utopies, comme les ordinateurs vêtements ("wearable computers" dans la terminologie anglo-saxonne), l'informatique embarquée — dans les voitures, dans les implants cardiaques par exemple.

Ainsi, l'une des raisons du succès de la carte à puce est qu'il se trouve qu'elle incarne précisément une solution aux exigences de mobilité, d'omniprésence et de support digital. La carte à puce est un dispositif d'archivage et de calcul, portable, personnalisé et fournissant un niveau de sécurité élevé.

Organisation du chapitre La section 2.2 introduit un court historique de la carte à puce. La section 2.3 propose une description plus technique de la carte à puce. La section 2.4 offre une description de la plate-forme JavaCard. La section 2.5 propose une réflexion sur l'utilisation des cartes à puce sous l'angle de la garantie de sécurité.

2.2 L'utilisation des cartes à puce

Nous rapportons dans cette section un rapide survol de l'évolution des cartes à puce et des applications auxquelles elles ont servi de support. Nous soulignerons ainsi comment la demande du marché dans lequel se place la carte à puce et les possibilités de la carte à puce elle-même se sont mutuellement stimulées.

La préhistoire En 1950, le Dinner's Club (USA) produisit les premières cartes à usage bancaire. Bien que très rudimentaires, elles permettaient le payement sans intervention d'argent "palpable". En 1968, Dethloff & Grötrupp (Allemagne) ont déposé le premier brevet sur un support plastique de la taille d'une carte bancaire contenant un dispositif électronique. C'est l'ancêtre de la carte à puce. Indépendamment, le même dispositif a été mis au point par Arimura (Japon) en 1970. Mais le coup de pouce décisif a été donné par Roland Moreno (France) en 1974, avec le dépôt de plusieurs brevets (47 brevets dans 11 pays de 1974 à 1978) donnant définitivement naissance à la carte à puce. Et dès la fin des années 70 les premières cartes à puce sont produites par CII-Honeywell-Bull (le future Bull). L'architecture physique des cartes n'a cessé d'évoluer depuis.

Les premières cartes à puce Si, initialement, Moreno visait, par la carte à puce, la création "des systèmes d'information de poche", le projet fut jugé trop avant-gardiste et donc inapplicable tel quel par les entités bancaires. Ce sont, en fait, les organismes de télécommunication qui ont compris l'opportunité que représentait les cartes et qui ont donc saisi les premiers l'occasion. Au début des années 80 apparaît la GeldKarte en Allemagne et la Télécarte en France. Le franc succès de la carte à jetons pour la téléphonie publique précède ainsi la fin des réticences des entités bancaires. Les capacités de stockage alliées à celles de chiffrement de la carte à puce ont finalement convaincu le GIE (Groupe d'Intérêt Économique) Cartes Bancaire qui voit là un moyen de réduire les fraudes et de proposer de nouveaux services. Toute utilisation de la carte, et c'est dorénavant une invariance dans son utilisation, implique l'introduction d'un code secret identifiant (PIN, pour Personal Identification Number). Le GIE Cartes Bancaire proposa ainsi avant la fin de la première moitié des années 80, les premières cartes à puce bancaires.

Vers une adoption généralisée de la carte à puce Après les succès des cartes à puce comme télécartes et cartes bancaires, la carte à puce s'ouvre à de nouveaux marchés. Se distinguent alors deux familles d'applications pour carte à puce. Les cartes à vocation d'archivage et les cartes à vocation applicative. Toutes deux impliquent une architecture logicielle différente. Les première sont des cartes incluant un SGBD (Système de Gestion de Bases de Données) et acceptant des requêtes du type SQL (Structured Query Language) [56]. Les deuxièmes, orientées traitement, incluent une ou plusieurs applications et un système de fichiers archivant les données nécessaires à celles-ci.

Cependant, si les cartes à puce sont bien admises, elles ne répondent pas au mieux aux contraintes des nouveaux marchés. Nous allons décrire dans ce qui suit comment les nouveaux champs d'applications ont amené à repenser l'architecture logicielle de la carte.

Les cartes santé qui sont des cartes à puce utilisées comme vecteur de transaction et d'information entre les différents organismes de santé, les assurances et le porteur ont commencé à fleurir dans les années 90. Par exemple, le GIE Sésame Vitale établit en 1996 un cahier des charges pour utiliser les cartes à puce comme carte santé. Une difficulté soulevée par les cartes à puce dans le cadre de cette utilisation est le délai de production des cartes qui est jugé trop long. Le constat est que le schéma jusqu'alors utilisé "un besoin = une architecture logicielle " est trop coûteux et trop peu générique. La solution apportée qui préfigurera les architectures ouvertes, préconise l'utilisation de cartes sur lesquelles il est possible de spécifier les données après production de la carte.

Les cartes GSM/SIM Mais c'est le domaine de la téléphonie mobile qui imposa définitivement la définition d'un nouveau modèle logiciel pour la carte à puce. Rappelons les enjeux du domaine. Outre le client, il y a deux acteurs impliqués dans la téléphonie mobile. Le premier, le constructeur, produit le téléphone et y encode même certains services d'aide à la communication (comme les agendas). Le deuxième, l'opérateur, fournit l'infrastructure pour relayer les communications ainsi que l'accès du téléphone à ce relais et, éventuellement, divers services (répondeur, achat en ligne etc...). Le service de base proposé par l'opérateur est proche de celui que propose les télécartes. De plus, pour des raisons évidentes l'opérateur ne veut pas dépendre du matériel et donc que ce service soit physiquement placé dans le téléphone. La carte à puce est donc une solution de choix pour l'opérateur. L'antagonisme entre opérateur de télécommunication mobile et constructeur de téléphones d'une part et l'évolution spectaculaire du matériel comme des services proposés d'autre part a obligé les deux protagonistes à établir des normes d'utilisation (GSM), comme aussi à définir une nouvelle architecture logicielle évolutive capable d'une part de charger et de faire cohabiter plusieurs services (SIM) et d'autre part de permettre de minimiser les coûts et temps de production. C'est cette capacité à l'évolution, d'abstraction du matériel et à la cohabitation qui a donné origine à la dénomination de *cartes à puce ouverte*.

Les Normes comme vecteurs de maturité Entre-temps, pour des raisons semblables, plusieurs normes (voir section 2.3) voient le jour. Apparaissent ainsi des normes standardisant l'interconnexion des cartes à puce avec différents environnements (avec les ordinateurs, les réseaux, etc), des normes établissant des protocoles de communication ainsi que des interfaces d'application (les APIs dans la terminologie anglaise) pour certaines familles d'applications.

Cette réflexion sur l'utilisation et le déploiement de la carte à puce en conjonction avec l'architecture ouverte des cartes, ne rendent compte que de la maturité du dispositif et préfigurent aussi bien la déclinaison de l'utilisation de la carte dans une multitude de domaines que les cartes à puce "tout-en-un" concentrant un ensemble de services qui nécessitaient plusieurs cartes auparavant.

Les cartes à puce ouvertes au présent En conséquence fleurissent les applications qui profitent des capacités d'archivage chiffré existantes pour utiliser les cartes comme identificateurs électronique (e-ID). De telles applications permettent l'utilisation de la carte, bien évidemment sous couvert du code secret PIN, comme certificat d'identité civile ou professionnelle, carte de contrôle d'accès, carte de contrôle d'utilisation de ressource (quelles qu'elles soient), comme signature digitale, ou encore comme support à la biométrie.

Citons aussi les applications reliées à la notion de "monnaie électronique". Les portefeuilles électronique (e-wallet ou e-purse selon la terminologie anglo-saxonne) qui sont promis à un bel avenir. Tout aussi doré semble le devenir des applications encartées supportant l'achat en ligne, le fameux *e-commerce*, ou *m-commerce* si celui-ci est réalisé via le téléphone portable. Dans ce groupe plaçons aussi les applications de fidélité (e-loyalty) qui permettent la collection et la gestion de points de fidélités acquis.

Pour finir la discussion sur les applications des cartes à puce, un autre domaine prometteur d'application des cartes à puce est sans aucun doute l'administration publique électronique (l'e-gouvernement, consulter [158]). Par là, nous entendons toute solution électronique aidant l'accomplissement des devoirs et droits du citoyen, comme par exemple le vote électronique (l'e-vote). Il est certain que la carte à puce a un rôle prédominant à jouer dans ce cadre.

Ce qui est notable dans cette foison de *e-applications*, et le format de la liste des applications que nous avons présentée est, dans ce sens, intentionnellement fastidieux, est l'engouement de l'industrie et des administrations publiques.

Nous sommes bien près de voir, après l'effet “dotcom”, l'effet “e-solution”. Espérons le avec un dénouement plus heureux.

2.3 Qu'est ce qu'une carte à puce ?

Cette section a pour objectif de donner un rapide aperçu de l'architecture physique et logicielle des cartes à puce.

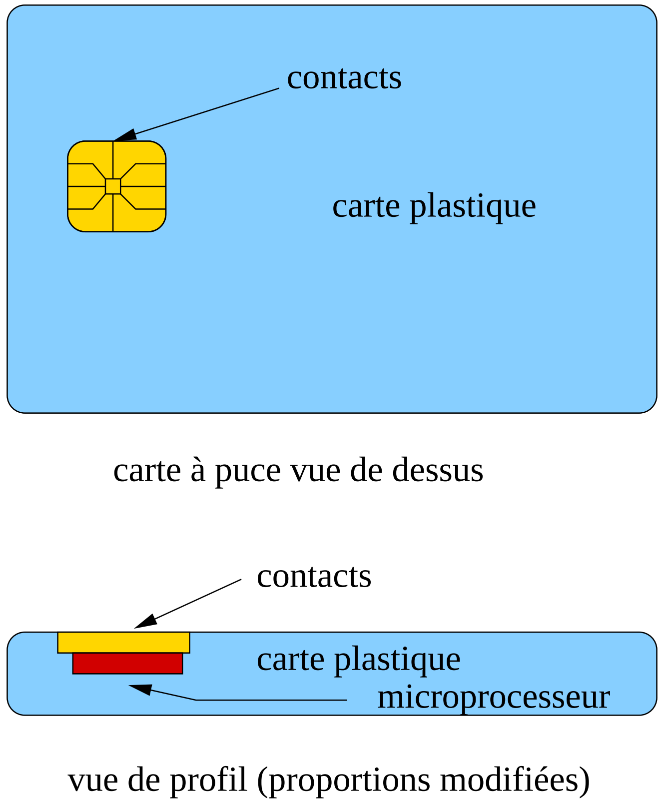

FIG. 2.1 – Vues d'une carte à puce au format carte bancaire.

Le format Une carte à puce est un micro-module (le plus souvent au format carte bancaire, $h : 54\,mm \times l : 85,6\,mm \times p : 0,76\,mm$) contenant un microprocesseur et une surface pour les contacts électriques (figure 2.1). Les contraintes de taille imposées à la carte à puce entraînent une architecture minimaliste. D'une part la carte à puce actuelle ne possède pas sa propre alimentation électrique ni ne possède de dispositif d'entrées-sorties. L'utilisation d'une carte à puce suppose donc l'utilisation d'une source d'énergie et d'un terminal de communication. C'est le rôle du terminal (ou CAD, pour Card Acceptance Device) de fournir cette interface. Bien qu'il existe des cartes communicant par ondes électromagnétiques (et ne nécessitant donc pas de contact physique pour qu'ait lieu l'échange de données), nous nous concentrons sur la famille des cartes à puce, plus nombreuses, nécessitant un contact physique avec un lecteur.

Le microprocesseur. La figure 2.2 schématise l'architecture monolithique standard des microprocesseurs pour carte à puce. Les plus communs sont des processeurs CISC (Complex Instruction Set Computing) avec un bus dont la capacité est de 8 bits et disposant d'une fréquence de 4,77 Mhz. Mais cela peut varier et on trouve des architectures RISC (Reduced Instruction Set Computing), des bus de 32 bits et des vitesses approchant les 40 Mhz. La taille maximale des microprocesseurs est, quant à elle, normalisée à $25\,mm^2$.

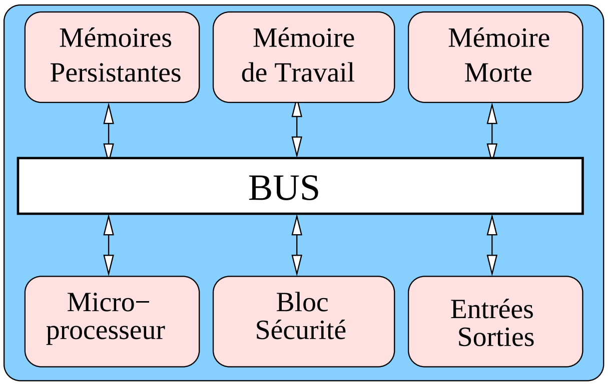

FIG. 2.2 – Architecture du microprocesseur d'une carte à puce.

Les cartes à puce peuvent disposer d'un coprocesseur conçu pour seconder le processeur dans les calculs numériques propres par exemple aux opérations cryptographiques. Le Bloc Sécurité de la figure 2.2 contient différents mécanismes de sécurité matériels comme, par exemple, des capteurs.

Les mémoires. Les cartes à puces contiennent trois types de mémoire.

La mémoire morte (ou encore ROM, pour Read Only Memory) est utilisée, de part sa nature persistante, pour stocker les informations qui sont fixées une fois pour toutes dans la carte. Cela inclut le système d'exploitation, les données permanentes et les applications de base. Le processus de *masquage*, qui est le processus durant lequel sont gravées ces données, a lieu pendant la phase de fabrication du microprocesseur.

La mémoire persistante (ou encore EEPROM, pour Electrical Erasable Programmable Read Only Memory) peut conserver, comme la ROM, les données stockées sans nécessiter d'alimentation électrique. Mais elle est de plus modifiable. Son utilisation a cependant un coût (le nombre de cycles d'écriture, la granularité — de laquelle dépend la taille et le temps de réponse à l'écriture) et se détériore au long des utilisations. Différentes technologies d'EEPROM proposant des variantes aux caractéristiques plus ou moins avantageuses tentent de palier à ces défauts. Les mémoires Flash semblent, par exemple, devenir une alternative de plus en plus viable aux technologies EEPROM actuelles.

La mémoire de travail (ou encore RAM, pour Random Access Memory) est utilisée comme espace de stockage pour les données temporaires et modifiables. Cette mémoire étant volatile ne conserve pas les données lors de perte d'alimentation. Elle a cependant l'avantage d'être rapide (tant à l'écriture qu'à la lecture) et de ne pas dégénérer, mais dispose d'une granularité interdisant l'existence de grandes quantités de RAM sur une carte à puce.

Ainsi, une des gageures de la programmation pour carte à puce est exactement d'utiliser au mieux, au vu de l'application conçue, les différentes mémoires.

Les cartes haut de gammes disposent de 64 Ko de ROM, 64 Ko de EEPROM et de 4 Ko de RAM.

Le modèle de communication Comme dans le cas des liaisons de chemins de fer à une seule voie, la communication entre la carte à puce et le terminal se fait dans les deux sens, mais ne se fait que dans un seul sens à la fois. On dit de tels protocoles de communication qu'ils sont *half duplex*. Au niveau des cartes à puce, la transmission de l'information a lieu par paquets, les APDUs (pour Application Protocol Data Unit) en suivant un protocole appelé TPDU (pour Transport Protocol Data Unit). Nous ne détaillerons pas le protocole mais en résumons son mode de fonctionnement. Pour commencer, il y a deux types d'APDUs qui correspondent aux requêtes et aux réponses à ces dernières : les APDUs de commandes et les APDUs réponses. Chaque type d'APDU a sa structure interne. En résumé une APDU de commande indique à quelle application elle se destine, quelle opération de l'application elle requiert avec, éventuellement, les paramètres nécessaires à cette opération. La réponse contient le résultat (qui peut être une erreur) de la requête. On peut distinguer des APDUs, ceux qui permettent la mise en place du protocole de communication, comme le signal de d'initialisation (ATR, pour Answer To Reset). La figure 2.3 illustre un scénario de communication où :

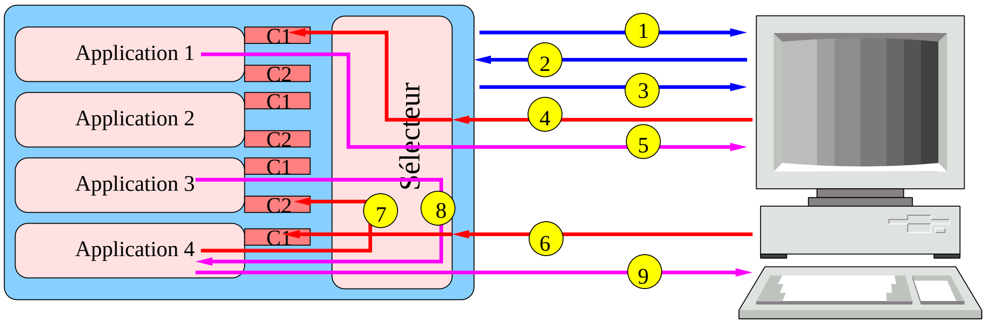

FIG. 2.3 – Communication par APDU.

1. la carte est mise sous tension, elle envoie donc un message d'initialisation au terminal (ATR) ;
2. le terminal négocie les termes du protocole ;
3. la carte répond et conclut la négociation ;

4. le terminal envoie une APDU de commande destinée à l'opération $C1$ de l'application 1. Le système d'exploitation de la carte, par son sélecteur, active l'application 1 et lui achemine l'APDU ;
5. Celle-ci calcule le résultat et envoie au terminal l'APDU réponse correspondante ;
6. le terminal envoie une APDU de commande destinée à l'opération $C1$ de l'application 4. Le sélecteur active l'application 4 et lui achemine l'APDU ;
7. l'application 4 traite de l'opération correspondante qui nécessite le service $C2$ de l'application 3. Une APDU commande correspondante à cette requête est préparée par l'application 4 et envoyé par le sélecteur à l'application 3 ;
8. l'application 3 retourne l'APDU réponse qui est acheminée à l'application 4 ;
9. Celle-ci répond finalement au terminal.

Le système d'exploitation natif Le système d'exploitation natif des cartes à puce est un petit système construit autour d'un système de fichiers hiérarchique élémentaire et qui repose sur les APDUs pour toute commande, en particulier celles de manipulation de fichier.

2.3.1 Les différents standards

Les standards statuant sur l'utilisation de la carte à puce ont joué, et continuent de jouer un rôle moteur dans le déploiement de cette technologie. Nous allons ici citer très brièvement les principaux standards et rappeler en particulier sur quels aspects de la carte à puce ils incident.

ISO 7816 Ce standard est le standard “fondateur” des cartes à puce. Le descriptif de la carte à puce qui précède est, par exemple, le résultat de ce standard. C'est aussi ce standard qui définit les contraintes de torsion que doivent supporter les cartes (et ce qui explique aussi la petite taille imposée à la puce). Il est composé de 7 parties. Citons, en guise d'illustration, la dernière partie qui traite du SCQL, la variante de SQL adaptée aux cartes à puce.

GSM L'un des plus franc succès de la carte à puce est son utilisation dans les télécommunications. Ce standard, publié par l'ETSI (European Telecommunications Standards Insitute), régit l'utilisation de la carte à puce dans le cadre de la téléphonie publique et mobile. Ce standard est aussi composé de plusieurs parties, dont une, par exemple, le GSM-03.19-SIM API, définit les APIs Java standard pour la programmation des cartes SIM.

EMV Un autre domaine d'application de choix des cartes à puce est la monétique. Reposant sur l'ISO 7816, le standard EMV (pour Europay, MasterCard et Visa, les organismes fondateurs de ce standard) statut sur les caractéristiques techniques supplémentaires que doivent posséder les cartes (et les infrastructures d'accueil, comme les terminaux) afin que celles-ci puissent servir de support aux applications bancaires.

OpenCard Framework (OCF) Initié par IBM, puis repris par le consortium OpenCard, ce standard est une réponse aux problèmes de dépendance entre les différentes parties composant l'infrastructure d'utilisation des cartes à puce (la carte, le lecteur de carte, le terminal, le système d'exploitation). Ce standard définit donc une interface pour l'interaction entre ces différentes parties. Là aussi c'est le langage Java qui a servi, à l'origine, de langage de support (à cette interface correspond donc une API Java).

PC/SC (pour Interoperability Specification for ICCs and Personal Computer System) propose une infrastructure logicielle pour l'utilisation des cartes à puce avec un ordinateur personnel. Il reprend (et étend) les choix et les solutions du standard OpenCard en prenant en compte les spécificités du matériel et des systèmes d'exploitation pour ordinateur personnel.

Open Platform La plate-forme ouverte (Open Plateform dans sa terminologie originale), développé à l'origine par Visa, définit un environnement intégré pour le développement et le déploiement des systèmes d'exploitation multi-applicatifs. Elle consiste en une spécification traitant de la carte et d'une spécification traitant des terminaux. Ceux-ci étendent et restent compatibles avec les différents standards cités. De tels systèmes ont la particularité d'accepter :
- la cohabitation de plusieurs applications sur la même carte ;
- le chargement d'une application après le déploiement de la carte ;
- l'interopérabilité des applications présentes sur la carte.

2.3.2 Les systèmes d'exploitation

L'avènement des plates-formes ouvertes préfigura l'apparition de plusieurs systèmes d'exploitation multi-applicatifs avec interopérabilité des applications. Citons les principaux : MULTOS du consortium MAOSCO [121], Smart Card For Windows de Microsoft [143], Camille de l'équipe RD2P de L'université de Lille [38], .NET Card de SchlumbergerSema [123] et bien sûr JavaCard [91] qui est au centre de cette thèse.

Avant de laisser place à une description des fondements de l'architecture JavaCard, il est intéressant de rappeler ici ce que Gilles Grimaud définit dans sa thèse [80] comme étant le cahier des charges d'un système d'exploitation pour carte à puce satisfaisant :

portabilité des programmes sur les différents microprocesseurs ;

compacité des structures de données et des traitements encartés ;

sécurité-innocuité des applications et de leurs données ;

hétérogénéité des langages de programmation utilisables ;

interopérabilité des programmes dans la carte et entre la carte et l'extérieur ;

flexibilité/extensibilité du système et des abstractions proposées ;

efficacité d'exécution des programmes.

Soulignons que les systèmes ouverts, par définition, offrent une réponse aux trois premiers points : portabilité, compacité et sécurité-innocuité. Dans la section suivante, après avoir décrit JavaCard, nous analyserons l'adéquation des réponses proposées.

2.4 La plate-forme JavaCard

La plate-forme JavaCard, l'un des premiers systèmes ouverts pour cartes à puce, a été originalement pensée et développée par les laboratoires de Schlumberger (en 1996) qui étaient à la recherche d'une architecture pour carte à puce fiable. Ils se sont naturellement tournés vers Java et son adaptation aux contraintes propres à la carte à puce. En effet Java proposait, dans son domaine d'application, les solutions dont ils voulaient disposer pour la carte à puce. SUN reprit le flambeau et établit les premières spécifications. Très rapidement les industriels de la carte à puce en ont saisi les enjeux et forment en 1998 le JavaCard Forum désormais promoteur de la technologie JavaCard. Nous traitons dans cette thèse de la version 2.1 qui est spécifié dans les documents [150, 151, 152].

Les technologies Java pour carte à puce bénéficient donc de l'appui d'une part des principaux acteurs de la carte à puce[1] et d'autre part de nombreux standards suggérant l'utilisation de ces technologies (GSM/SIM, OCF en sont des exemples). Cet état de fait confère à JavaCard une pérennité et un certain "capital confiance". De facto la plate-forme JavaCard prend une place prépondérante dans la multitude des systèmes d'exploitation pour carte bancaire.

Java en deux mots La plate-forme Java [90] est composé d'un langage orienté objet fortement typé à héritage de classe simple et à héritage d'interface multiple. C'est aussi un langage multithreadé, disposant d'un mécanisme de gestion d'exceptions et dont le code compilé, appelé bytecode, est destiné à être interprété par une autre composante de la plate-forme Java : la Java Virtual Machine (JVM) qui est une machine virtuelle à pile. Dans la terminologie Java, une application est appelée applet. Pour finir ce rapide descriptif du langage, soulignons qu'il ne fournit pas de mécanisme direct pour la manipulation de pointeur mais qu'il dispose cependant d'un ramasse miettes permettant de récupérer les cellules non utilisées. Nous reviendrons sur l'importance du ramasse miettes pour JavaCard section 2.5.

De Java à JavaCard L'adaptation de la technologie Java aux cartes à puce doit essentiellement prendre en compte l'architecture d'accueil fortement contrainte. Si la plupart des concepts de Java sont transposés dans JavaCard, d'autres ont dû être modifiés ou être simplement omis. Les spécifications SUN [150, 151, 152] et [43] introduisent en détails les principales caractéristiques de

[1]Nous avons, à ce propos déjà cité les industriels de la carte à puce, mais nous trouvons aussi le secteur de la télécommunication et le secteur bancaire qui passe progressivement mais massivement de MULTOS à JavaCard .

JavaCard et de son héritage Java. Nous n'aborderons ici que les traits les plus caractéristiques et pertinents à notre propos.

Un comparatif entre les langages Java et JavaCard peut être résumé par le tableau suivant :

Caractéristiques Java supportées	Caractéristiques Java non supportées
Les types primitifs de petite taille : `boolean`, `bytes` et `short`. le type `int` est optionnel	Les autre types primitifs (de grande tailles), les caractères et les chaînes.
Les tableaux à une dimension	Les tableaux à plus d'une dimension
Les packages, les classes, les interfaces, les exceptions	Le chargement dynamique de classe
L'héritage, les méthodes virtuelles, la surcharge, la création dynamique d'objet	Le gestionnaire de sécurité
Le gestionnaire d'exceptions	Le ramasse miettes, et la finalisation
	Le multithreading
	La sérialisation
	Le clonage

Sur le plan de l'architecture logicielle, la carte à puce impose certaines adaptations :

Le format CAP : Le code encarté est condamné à être compact. Dans ce sens, la technologie JavaCard fournit un module permettant une transformation, optimisation et simplification des fichiers CLASS (issus de la compilation Java). La *conversion*, qui est l'étape de transformation des fichiers CLASS, comprend par exemple une étape dite de *tokenisation* où les noms sont remplacés par des *tokens* (moins encombrants, plus compacts) ou encore une édition de lien simplificatrice. Le nouveau format, nommé CAP pour *Converted APplet*, est le format des applets encartées et est directement exécutable par la machine virtuelle [152] ;

Le vérifieur de bytecode : Dans l'architecture Java, le chargement dynamique de classe s'accompagne d'une étape de vérification (du typage, de la sécurité/innocuité). Même s'il n'y a pas ici de chargement dynamique, il existe la possibilité de charger du code après distribution de la carte. Une étape de vérification reste donc néanmoins nécessaire, mais jusqu'à assez récemment les différents scénari d'installation d'applet ne comprenaient pas la vérification sur la carte elle-même (consulter par exemple [39, 106, 107] pour plus de précisions) et lorsque que le schéma choisi comprenait une étape de BCV, celle-ci n'était possible qu'en dehors de la carte, au moment de la conversion CLASS-CAP.

Ce dernier point est particulièrement important. Il sous-tend, comme nous le verrons plus loin, une grande partie de la problématique discutée dans cette thèse (consulter par exemple les sections 2.5 et 4.8).

La plate-forme JavaCard (figure 2.4) se compose, sur la carte, du JCRE (l'environnement d'exécution de JavaCard) qui se place au dessus de la couche logicielle native de la carte à puce et au dessous des applications. Le JCRE dont le but est d'organiser l'exécution des applications est composé principalement des APIs standard et de la machine virtuelle (JCVM). Celle-ci contient entre autres les modules propres à la gestion des exceptions, de la mémoire et du pare-feu.

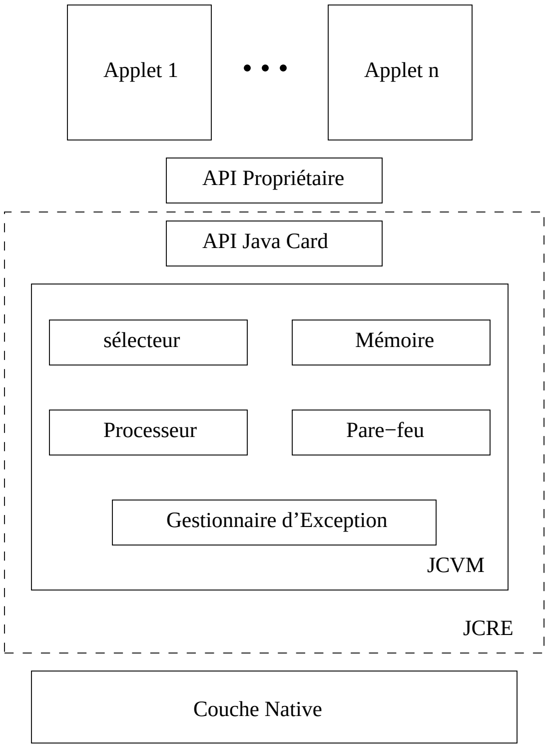

FIG. 2.4 – Structure logicielle JavaCard sur la carte à puce.

Cycle de prodution d'une applet JavaCard La figure 2.5 schématise le cycle de production d'une application JavaCard. Une applet JavaCard est écrite dans le langage Java mais devra se limiter aux constructions autorisées par JavaCard. Il n'utilisera, de plus, que les APIs destinées à JavaCard (celles de la spécification originale, ou celles, par exemple, spécifiques aux applications GSM, etc.). La compilation, standard, par un compilateur java produit les fichiers CLASS correspondants. Vient ensuite la phase de conversion. La conversion pourra inclure une phase de vérification (par le BCV). À ce moment l'applet peut être installée dans la carte et est, ensuite, en mesure d'être exécutée.

Les objets transients Le modèle mémoire distingue deux types d'objets, les objets persistants et les objets transients. Les objets persistants conservent leur valeur au gré des différentes sessions, ils sont stockés dans la mémoire persistante (EEPROM). Les objets transients sont des objets dont les données sont volatiles. En résumé, il existe plusieurs types d'objets transients qui varient selon les

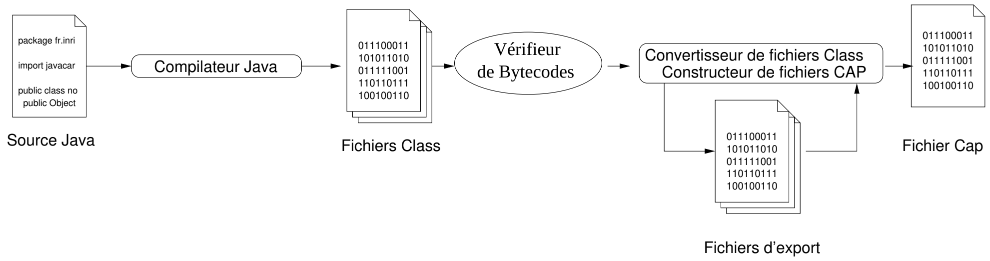

FIG. 2.5 – Compilation d'un programme JavaCard.

modalités de "volatilité". Cette distinction suggère que ces objets soient stockés en RAM et donc qu'ils soient adaptés au stockage des données sujettes à modifications fréquentes ou d'utilisation temporaire. L'existence de plusieurs types d'objets induit un coût. D'une part ils révèlent un manque d'abstraction certain par rapport à l'architecture native. D'autre part nous savons déjà qu'une utilisation non parcimonieuse de tels objets — comme soumettre de fréquentes mise à jour à un objet placé dans la mémoire persistante, a une incidence négative sur la durée de vie de la mémoire de la carte.

Les transactions Par le biais du système d'exploitation natif, JavaCard assure l'atomicité d'une opération d'écriture de base (l'écriture d'un champ par exemple). Il peut être cependant nécessaire de garantir non pas l'atomicité d'une opération mais l'atomicité groupée d'un ensemble d'opérations (ou toutes les opérations sont exécutées ou aucune d'elles ne l'est). Ceci est particulièrement important dans un cadre où la carte peut être retirée à tout moment du lecteur et compromettre l'état des données. Dans la terminologie JavaCard, les transactions sont ces ensembles d'opérations atomiques. Le système d'exploitation natif assure cette atomicité. Au niveau de JavaCard, c'est le JCRE qui est charger d'orchestrer les demandes de transaction et de servir d'interface avec la couche native.

Le modèle de sécurité de JavaCard JavaCard hérite de Java des mécanismes de modificateurs d'accès qui régissent la visibilité des méthodes. D'autre part, nous avons déjà abordé le statut particulier du BCV dans la plate-forme JavaCard. S'il n'y a pas de gestionnaire de sécurité à la Java (bac à sable, contrôle d'accès), la plate-forme dispose d'autres mécanismes plus adaptés aux architectures ouvertes. C'est le rôle du pare-feu, aux ordres du JCRE, d'actionner ceux-ci :

– le cloisonnement des applications par contextes : c'est l'équivalent JavaCard du bac à sable. Afin d'éviter toute intrusion inopinée d'une applet dans le "terrain de jeux" d'une autre applet, JavaCard définit des cloisons— les contextes dans la terminologie JavaCard, dans lesquels toute applet à droit de "jouer" mais en dehors de laquelle tout contact sans supervision du JCRE est interdite. La topologie du contexte est celle du package : deux applets définies dans le même package appartiennent au

même contexte ;
- le partage d'objet : le cloisonnement est une mesure nécessaire à la garantie de l'innocuité du code, mais est nuisible à l'interopérabilité des applications. Par un protocole reposant sur une interface appelée `Shareable` et pilotée par le JCRE la communication inter-contextes est possible. Un objet peut ainsi partager ses ressources de façon contrôlée avec une applet n'appartenant pas à son contexte.
- les objets point d'entrée et les tableaux globaux sont des éléments de l'architecture JavaCard appartenant au JCRE et qui sont partagés (accessibles) sans restriction. Ils échappent ainsi aux politiques restrictives de cloisonnement et de partage contrôlé. Ces entités sont les services systèmes que le JCRE fournit à toute applet (services de communication, requête de service, tampon APDU etc...)

En guise de conclusion à cette succincte présentation, rapprochons JavaCard des critères énoncés section 2.3.2. Si l'architecture ouverte de JavaCard apporte une solution satisfaisante aux deux premiers critères (portabilité, compacité) il n'en va pas de même pour les autres. La solution que JavaCard apporte au problème de la sécurité/innocuité est incomplète. Pour commencer, la sécurité/innocuité est contradictoire avec l'interopérabilité. Trop favoriser l'un c'est aller à l'encontre de l'autre. JavaCard propose un compromis qui n'est pas tout à fait satisfaisant comme nous le verrons dans au paragraphe intitulé *"la couche logicielle"*, page 45. Sur ce point encore, ajoutons que fleurissent actuellement de nouveaux scénari d'utilisation de BCV. Avec ceux-ci, il deviendrait enfin envisageable de charger un BCV sur la carte, ce qui améliorerait de façon drastique l'architecture sécuritaire de JavaCard. Sur un autre plan, l'ingénierie du matériel propose des solutions de plus en plus performantes et les contraintes des cartes à puce sont sans cesse repoussées : des mémoires de meilleure qualité et au grain plus fins, des microprocesseurs plus puissants. En parallèle à cette évolution, JavaCard gagne en maturité. En conséquence, (1) on peut espérer voir se définir une abstraction du matériel (particulièrement au niveau de la mémoire) et des couches natives (notamment au niveau des communications—par APDU) plus satisfaisante ; (2) il est maintenant envisagé d'inclure officiellement dans les prochaines version de la plate-forme des fonctionnalités propres à Java. Le multithreading et le ramasse miettes en sont des exemples.

L'hétérogénéité est l'un des points forts de l'architecture Java[2], il est tout a fait raisonnable de penser qu'il le sera aussi pour JavaCard. Considérons par exemple les efforts de [141].

2.5 Les cartes à puce sont-elles le Saint Gräal de la sûreté ?

Répondons tout suite à cette question par la négative. Il n'existe pas de système 100% fiable. Mais il est important de connaître les risques afin de mieux

[2] En fait, l'hétérogénéité d'une plate-forme trouve sa source dans les spécifications complètes et publiques. Ce qui est le cas de JavaCard.

les minimiser (contrôler), de connaître les limites d'un système afin de mieux les repousser.

2.5.1 La sécurité comme un processus complexe

B. Schneier [139] parle de la sécurité d'un système comme étant non pas un produit, mais un processus. Dans ce sens il déclare :

> *la sécurité est une chaîne dont la résistance est celle du maillon le plus faible*

C'est dans cette perspective que nous aborderons dans cette section, sans pour autant chercher à être exhaustif, la résistance de la chaîne sécuritaire des cartes à puce et de JavaCard. En particulier nous montrerons quelques chaînons problématiques. Une telle approche est classique, citons, entres autres, [6, 7, 8, 71, 75, 111, 116]. Ce constat nous amènera à réfléchir et à justifier l'intérêt de la vérification formelle de la plate-forme JavaCard.

Déploiement des cartes. Avant d'atterrir dans les mains de son propriétaire, un carte à puce suit tout un chemin semé d'embûches. De même l'utilisation au quotidien de la carte est loin d'être, sur le plan de la sécurité, anodine.

L'émission des cartes et la gestion des données s'y rapportant (comme l'attribution et la communication du PIN) est un processus sensible. Les institutions bancaires, pour ne parler que de ce cadre, ne l'ont pas forcément toujours contrôlé de la meilleure façon. L'article [6] rapporte de nombreuses anomalies liées à la gestion des cartes bancaires au Royaume Uni. Citons, par exemple, le cas d'un employé de banque ayant commandé, au nom d'un client et à son insu, une carte bancaire supplémentaire vers une adresse dont l'employé avait le contrôle. L'ironie de la situation est que le relevé des transactions effectuées par cette deuxième carte allait aussi à l'adresse indiquée. Ceci a, bien évidemment, difficulté la découverte de la fraude par le véritable propriétaire. L'éviction de telles anomalies, lorsqu'elles sont découvertes, est néanmoins relativement aisée, ce qui est le cas de l'exemple cité.

Néanmoins, un état de fait assez déroutant, que l'on retrouve fréquemment au niveau de ces institutions, est la tendance à reposer leur politique de sécurité sur la *sécurité par l'obscurité*. Déroutant car d'une part, être dans le secret ouvre de véritables autoroutes aux esprits malins et d'autre part, celles-ci, lorsqu'elles sont prises en défaut, ont tendance à nier et à ne pas être prompt à la rectification qui leur serait pourtant providentielle.

R. Anderson, toujours dans [6], rapporte les risques propres à l'utilisation quotidienne des cartes. Il est assez commun de garder son code secret dans son portefeuille, d'entrer son code sans précaution dans le lecteur de carte d'un magasin, etc.

Le terminal et la communication. Un terminal doit être digne de confiance et résistant aux attaques. Tel est le cahier des charges d'un ATM sur le plan de la sécurité. Dans l'optique du porteur de la carte, l'utilisation du terminal est cependant une question de confiance. Or de nombreuses malversations ont été rapportées, tant au niveau du terminal lui-même que dans les échanges entre carte et terminal. Citons quelques exemples classiques du *problème de terminal* :

- les faux terminaux. On peut raisonnablement penser qu'un terminal placé auprès d'une banque ne soit pas une falsification, mais quelles garanties avons nous avec les autres terminaux, les lecteurs de carte dans les magasins ? Actuellement, des solutions basées sur l'authentification mutuelle (de la carte et du terminal) apportent une réponse à ce type de problèmes ;
- les terminaux défectueux ou mal programmés. Un cas paradigmatique est celui de l'ATM qui réactivait la dernière session lorsqu'une carte de téléphone était insérée (voir [6]) ;
- les terminaux pris en otage par malveillance. Des fraudes perpétrées par des agents de maintenance des terminaux ont déjà été signalées. De plus certains terminaux sont en fait des ordinateurs personnels avec lecteur de carte. Or, ceux-ci ne sont pas réputés pour être des citadelles inviolables.

Le deuxième point concerne les protocoles de communication, que celle-ci soit entre la carte et le terminal ou interne à la carte. Un point délicat et bien connu des experts en analyse de protocoles est qu'il est vraiment très ardu de concevoir et de valider un protocole solide comme il est très difficile de faire interagir sans risque plusieurs protocoles qui individuellement ne posent pas de problèmes. Un exemple simple de la mise en défaut d'un protocole de communication interne est celui décrit par l'article [23] où une applet réussit à acquérir un service qui ne lui était pas destiné par la simple observation des applets ayant droit.

La carte est un dispositif physique. La carte est un dispositif électronique. Comme tel, il est possible d'observer, d'analyser et de quantifier, dans une certaine mesure et avec le matériel adéquat, les flux électriques la parcourant.

Les attaques physiques de la carte procèdent par, dans le jargon anglo-saxon, *reverse engineering* de la carte. Il est possible de déduire ce que contient la carte par analyse de son comportement électrique. Cette analyse a été démontrée réalisable en pratique sur les cartes à puce. Le tout est de posséder la carte et d'être en mesure de produire les stimuli compromettants.

Plusieurs techniques ont vu le jour, et sont particulièrement bien documentées. Citons deux exemples :

- l'analyse dite *Differential Fault Analysis* qui place la carte en situation extrême (par fluctuation de température, d'alimentation électrique, du signal d'horloge, par choc violent etc...) l'amenant à un état d'erreur. L'analyse du comportement de la carte permet de déduire ce que contient la carte. [7, 8] introduisent un autre type d'attaque ayant abouti par analyses variées des composants électroniques ;
- l'analyse dite *Differential Power Analysis* : celle-ci consiste à inférer statistiquement les données, à partir de l'analyse de la consommation électrique de la carte pendant une opération (via un ATM modifié), car il est possible d'établir une relation entre consommation et calcul.

Des mesures ont été prises pour faire face aux attaques connues (introduction de bruits aléatoires, un meilleur masquage des flux, etc...). Celles-ci ont été implémentées dans le Bloc Sécurité du microprocesseur de la carte à puce (se référer à la figure 2.2). Cependant, ces parades ne rendent pas impossible l'analyse, elles ne font qu'augmenter coût et temps nécessaires à celle-ci. La protection aux attaques de ce genre est impossible. Le but avoué des concepteurs de carte à puce est alors de pousser la résistance le plus loin possible pour que l'attaque soit de plus en plus coûteuse.

La couche logicielle. Jusqu'à maintenant nous avons insisté sur les attaques résultant sur la découverte du PIN. C'est l'objectif premier de tout attaquant. Cependant le code secret n'est pas la seule information profitable placée dans une carte à puce. Les exigences de nouvelles fonctionnalités qui ont conduit aux plates-formes ouvertes ont introduit de nouveaux mécanismes (cohabitation et chargement de plusieurs applets sur une même carte, principalement). Ceux-ci ont eu comme conséquence une complication nette du modèle de sécurité. Les risques encourus par la carte ne se confinent plus qu'à la découverte du PIN, mais couvrent maintenant le détournement de service et l'hébergement d'applets hostiles. Ce cas de figure est représentatif de la cohabitation entre les exigences fonctionnelles et les exigences de fiabilité. Les deux ne font pas toujours bon ménage, les avancées dans une direction compromettent souvent celles de l'autre. Java Card n'échappe pas à ce constat. Un point délicat à prendre en compte dans la solution, somme toute assez récente, de JavaCard est la relative immaturité des mécanismes impliqués. C'est en les éprouvant qu'apparaissent des faiblesses qui sont répertoriées et, heureusement, en mesure d'être corrigées.

Spécifications ambiguës : par expérience (voir chapitre 4), l'implémentation d'une plate-forme JavaCard à partir de spécifications informelles peut s'avérer déroutante. Des imprécisions ou des ambiguïtés peuvent mener à des implémentations aux comportements différents. Or [71] rapporte les dangers de tels écarts de comportements ;

Absence de ramasse miettes : les spécifications actuelles de la plate-forme de JavaCard ne comprennent pas de ramasse miettes ni ne fournissent au programmeur de moyen de libérer de la mémoire. Il n'est pas clair que l'on puisse éviter le scénario où une applet malveillante ou défaillante alloue de la mémoire jusqu'à exhaustion de la mémoire et blocage de la carte. Les concepteurs de carte à puce en ont, heureusement, saisis très tôt les conséquences. Ils incluent ainsi un ramasse miettes à leurs implémentations de la plate-forme JavaCard ;

Transgression du modèle de sécurité JavaCard : [116] rapporte une attaque du modèle de sécurité de JavaCard. Celle-ci consiste essentiellement, pour une applet et par une conversion judicieuse de référence (*cast* dans sa terminologie originale) à acquérir un service qui ne lui était pas à l'origine destiné. L'article cité propose un nouveau protocole de partage d'objet, reposant sur la délégation, qui corrige ce problème ;

Transitivité du flux d'information : sans que le mécanisme de partage d'objet soit pris en défaut, il existe des configurations où une applet capture une information qui ne lui était pas destinée. Si une applet A confie une information confidentielle à une applet B, rien n'empêche à B de la communiquer à C [75]. Le problème de transitivité du flux d'information, qui est le terme générique utilisé pour désigner cette classe de problèmes, n'est en fait pas du ressort de la plate-forme. Nous avons là un comportement que l'on aimerait voir évité au niveau des applications : ce n'est pas à la politique de sécurité de la plate-forme de prendre en compte cette configuration, mais bien à une politique de sécurité des applications chargées qui viendrait compléter celle du JCRE ;

Chargement d'applets après distribution de la carte : nous avons déjà mentionné à plusieurs reprises les problèmes de sécurité occasionnés par la capacité à charger de nouvelles applications sur la carte. Un chargement sans précaution est la porte ouverte à l'installation d'applets défectueuses, malveillantes et autre virus. Plusieurs protocoles de chargement ont été proposés, chacun avec ces contraintes propres. De récentes avancées laissent prévoir enfin une solution satisfaisante : la vérification de la sécurité/innocuité sur la carte elle-même [39, 42, 107] ;

Les méthodes natives sont une arme à double tranchant. Elles sont nécessaires à certains traitements. Malheureusement, sur le plan des mécanismes de sécurité, les méthodes natives échappent au contrôle du JCRE car elles s'adressent directement à la couche native. Une illustration simple est la méthode native demandant directement au système natif la valeur d'une cellule mémoire que le JCRE aurait protégé.

Pour finir, il est intéressant à ce niveau de faire une petite digression sur l'imaginaire distillé par certaines œuvres d'anticipations à propos des sociétés digitales globalisées. Si, parmi les grands classiques, le tableau dressé n'est pas franchement positif — prenons par exemple les sociétés "big-brother" de *1984* (Orwell, 1948) ou encore du *meilleur des mondes* (Huxley, 1932) pour ne citer que ceux-ci, c'est plutôt sur les dérives de son utilisation que sur la technologie elle-même que ceux-ci se prononcent. À contre-pied, *la nuit des temps* (Barjavel, écrit beaucoup plus tard, en 1968), l'œuvre qui aurait suscité l'idée des cartes à puce à Roland Moreno[3], décrit une de ces sociétés sous un jour plutôt naïf (voire même tendancieusement naïf) mais franchement positif. Dans cette société globalisée chaque citoyen possède sa véritable contrepartie numérique, une clé, qui est un dispositif utilisé pour toute transaction de la vie quotidienne (authentification, transactions financières, accès et régulation d'accès aux services dispensés par l'état) et même comme régulateur social : elle est utilisée pour désigner le partenaire parfait et pour servir de contraceptif !

Si ces œuvres nous ont mis en garde, elles nous ont surtout appris à être vigilant et à mettre les mauvaises utilisations de la technologie (les cartes à puce dans notre cas) au même rang que les problèmes de sécurité et de fiabilité[4], même si, rappelons le quand même, ce n'est pas dans cet esprit de contrôle et d'avilissement qu'ont été pensées et déployées les cartes à puce.

---

[3] Le conditionnel est ici employé pour souligner le coté anecdotique de cette affirmation. La "légende" *veut* que R. Moreno ait été inspiré par *la nuit des temps* alors que Moreno lui même s'en défend.

[4] On peut (doit) même considérer les méthodes formelles comme étant un des moyens disponibles pour exercer cette vigilance.

2.5.2 La vérification formelle de la plate-forme JavaCard est elle justifiée ?

Vérification de la plate-forme Force est de constater que la chaîne sécuritaire de la carte à puce n'est pas un modèle de solidité. Il peut sembler même désarmant au concepteur de plates-formes sécurisées pour carte à puce d'évoluer dans une tel contexte. Doit on pour autant jeter l'éponge ?

Sur certains chaînons la lutte peut sembler vaine. Se faire voler son PIN est imparable. On ne voit pas comment faire face au dysfonctionnements des organisations, des intervenant humains, ou encore des dysfonctionnements physiques (voir [6]).

Il n'en est heureusement pas de même pour la couche logicielle. La mise à l'épreuve d'une implémentation peut porter ses fruits. Même s'il convient de bien modérer et replacer dans son contexte les contributions que peuvent apporter les méthodes formelles, il ne fait pas de doute que celles-ci ont un rôle à jouer, ne serait-ce que par leurs bases mathématiques bien fondées et éprouvées.

Nous allons reprendre l'image de la sécurité comme un processus pour, comme nous l'avons annoncé, replacer la vérification formelle de la plate-forme au contexte auquel elle appartient. Le constat que nous aimerions dresser peut être formulé synthétiquement par :

la vérification formelle de la plate-forme n'est pas suffisante mais est nécessaire.

La vérification de la plate-forme n'est pas suffisante. En effet, au niveau *macroscopique*, la sécurité de la plate-forme est un maillon parmi d'autres. Garantir que la carte à puce est un dispositif sûr, c'est valider la totalité des maillons. Au niveau *microscopique*, la situation est tout aussi problématique. A. Ghosh [71] livre, dans le cadre de l'analyse des protocoles, la réflexion suivante :

> *Despite the amount of formality and rigor used in analysing protocols for security , it is clear that it is not possible to prove that any given protocol is secure. Part of the problem lies in defining what secure means. It is possible, for instance, to prove certain assertions about protocols [...], however, no general purpose method exists for determining how secure a given protocol is in general.*

Malheureusement ce constat se transpose dans toute sa puissance au cadre de l'analyse de tout système logiciel où la sécurité joue un rôle prépondérant. La plate-forme JavaCard n'y échappe pas. Définir et prouver la sécurité dans toute sa généralité est illusoire.

La vérification de la plateforme est nécessaire. Si les arguments de complétude dressent un tableau bien sombre, il est néanmoins important de rappeler que la vérification de la plate-forme est un élément du processus de sécurité et que, comme tel, mérite qu'on s'y attarde. La vérification de la plate-forme peut être considérée comme un minimum d'autant qu'elle a déjà permis la correction de nombreuses imperfections. Pour certains, la sécurité relative obtenue par une analyse formelle est déjà satisfaisante. Rappelons à ce propos les Critères Communs, que nous retrouverons plus en détail au chapitre 3, qui imposent la vérification formelle comme condition nécessaire à l'obtention du plus fort niveau de sécurité.

Pour finir, citons encore A. Ghosh :

> *In spite of these secure features, it is important to realize, however that much of the card security is based on two assumptions : the JavaCard Runtime Environment (JCRE) is correctly implemented, and that applets loaded on the card do not behave maliciously.*

Cette thèse propose un cadre à la vérification des deux hypothèses. D'une part valider la première hypothèse incriminée impose une vérification de la plateforme (chapitre 4), tandis que la deuxième hypothèse implique une vérification des applications elles même. Ce que suggère ce dernier point est, d'une part, l'obligation d'une étape de BCV, et les efforts de vérification sur la carte s'en retrouvent grandement justifiés, d'autre part la vérification formelle de programmes JavaCard, à laquelle nous proposons un support.

Le chapitre 3 est, à ce propos, est une introduction aux méthodes formelles, au contexte de la vérification formelle des plates-formes Java-JavaCard.

Chapitre 3

Les Méthodes Formelles

3.1 Cadre et motivation

Les systèmes d'information (SI) prennent une place de plus en plus prépondérante dans des secteurs essentiels des sociétés modernes. Les secteurs des transports, de la communication, de la santé ou de la production énergétique en sont des exemples. Paradoxalement, la croissance rapide en terme de complexité des systèmes informatiques qui accompagne leur utilisation et leur évolution, rend plus difficile la confiance que l'on peut leur porter. De fait, s'assurer que le système se comporte comme prévu est un problème de plus en plus épineux.

Sur le plan de l'utilisation, bien que la plupart des erreurs logicielles ne conduisent qu'à des inconvénients mineurs, la situation dans des systèmes dit *critiques* (les équipements nucléaires, ou équipements médicaux en sont des exemples classiques) est tout autre. Le comportement correct des composants logiciels de tels systèmes est essentiel. Soulignons, de plus, que l'émergence des nouvelles technologies et la révolution des technologies de l'information ne font que renforcer la nécessité de propriétés de correction.

La médiatisation des effets retentissant des erreurs de conception n'est pas, non plus, étrangère à cet engouement pour la fiabilité. Gouvernements, consortium industriels, ou tout simplement l'utilisateur de systèmes informatisés en ont assurément pris conscience et en exigent son établissement. Prenons un exemple de cette dissémination. La revue de vulgarisation scientifique, *Scientific American*, publiait en 1994 un article [72] déclarant :

> *Despite 50 years of progress, the software industry remains years – perhaps decades – short of the mature engineering discipline needed to meet the needs of an information-age society.*

L'industrie du logiciel traîne malheureusement depuis longtemps la réputation de manquer à ses engagements de fiabilité, de robustesse. De fait, la découverte de bogues dans les logiciels commerciaux est si habituelle que les compagnies mettent en place systématiquement un système de rapport de bogues à la sortie d'un de leurs logiciels.

Pour ne citer que des chiffres récents, l'institut nord-américain pour les standards et les technologies (NIST) estimait les coûts des bogues sur l'économie nord-américaine à 59 milliards de dollars en 2002 [1].

[1] http://www.nist.gov/public_affairs/releases/n02-10.htm

Si nous abandonnons le point de vue de l'utilisateur pour prendre celui du concepteur de systèmes d'information, la qualité du produit en terme de fiabilité devient aussi un enjeux commercial. Pour preuve, Goguen, dans [76], liste quelques contrats importants mis en péril par des considérations de fiabilité. Citons par exemple la résiliation du contrat de 8 milliards de dollars entre IBM et la US *Federal Aviation Agency* portant sur un système de contrôle aérien.

En 1994 a été découverte une erreur dans la division calculée par les processeurs Pentium. Bien que des millions de processeurs aient été déjà vendus, Intel a du rapatrier et échanger gratuitement toutes les unités produites [109]. Mais au delà de l'impact économique et financier, la presse mondiale a fini par souligner la perte de confiance que les utilisateurs ont manifestée auprès d'Intel.

Nous avons illustré au chapitre précédent combien il était difficile d'établir la garantie *absolue* de sécurité d'un système. Cependant, si la garantie est illusoire, le renforcement de la fiabilité et la confiance que l'on peut avoir en un produit continuent à être une préoccupation justifiée et centrale. Dans l'industrie, diverses mesures ont été mises en place pour affronter ce problème. La principale est le test et la simulation. Les méthodes formelles sont une approche qui retient de plus en plus l'attention des industriels.

La fiabilité par le test et la simulation Une approche classique employée pour renforcer la justesse d'un système est le *test* ou la *simulation*. Celle-ci consiste à fournir à une modélisation du système un ensemble de données jugées représentatives et de comparer la réponse avec le résultat attendu. Malheureusement il n'existe pas de test satisfaisant dans l'absolu. Pour l'écrasante majorité des systèmes, l'espace des valeurs possibles à tester est immense (lorsqu'il n'est pas infini), ce qui rend impraticable le test exhaustif. Une fiabilité relative est atteinte par un choix judicieux des valeurs d'entrées. Ces données doivent mettre à l'épreuve le plus de composants possibles du système testé. Là aussi, la complexité du choix des valeurs d'entrées est en commune mesure avec la complexité du système. Il devient ainsi difficile d'en garantir la robustesse. Or ce point est particulièrement important lorsqu'on parle de sécuriser un système embarqué. En effet la plupart des violations de sécurité sont provoquées par une utilisation inattendue des ressources.

Les méthodes formelles : une approche alternative ou complémentaire à la simulation Les *méthodes formelles* fournissent une approche mathématique différente à l'assurance de fiabilité. Au cœur de celles-ci se trouvent des formalismes et des outils mathématiques permettant la modélisation et le raisonnement sur les systèmes informatiques. J. Rushby [136] définit à ce propos les méthodes formelles par :

> *The term* formal methods *refers to the use of mathematical modeling, calculation and prediction in the specification, design, analysis and assurance of computer systems and software. The reason it is called* formal methods *rather than* mathematical modeling of software *is to highlight the character of the mathematics involved.*

Les méthodes formelles offrent enfin la possibilité de raisonner sur tout l'ensemble des données et des configurations possibles et non plus sur un échantillon. Bien qu'à l'origine elles aient été développées et utilisées dans le monde académique, les méthodes formelles sont de plus en plus fréquemment utilisées dans un contexte industriel, particulièrement dans le cadre des systèmes critiques. Le déploiement des méthodes formelles est conditionné par le coût qu'elles induisent : celui de l'expertise nécessaire à leur utilisation. Si le coût reste encore élevé, celui-ci ne cesse de baisser et le secteur industriel est de plus en plus enclin à le payer. D'une part les nécessités se font plus pressantes, d'autre part les méthodes formelles, de par leurs fondements rigoureux, apportent une solution jugée satisfaisante (voir section 3.2).

Les méthodes formelles pour les cartes à puce J. Harrison dans [84] rappela à quel point l'épisode du bogue du Pentium fut problématique pour Intel. Il souligna ainsi l'intérêt stratégique de la vérification formelle des microprocesseurs et son adoption par les laboratoires de recherche Intel. La situation est de fait identique pour les cartes à puce. Le marché que représente les cartes à puce est immense. Découvrir des dysfonctionnements de celles-ci après leur distribution serait catastrophique pour les industriels de la carte à puce.

Organisation du chapitre Le but de ce chapitre est d'introduire les méthodes formelles et leur utilisation à la vérification des plates-formes Java et JavaCard. Pour ce faire nous présentons une courte introduction aux Critères Communs dans la section 3.2. Cette norme justifie une partie des travaux de vérification formelle de plates-formes sécuritaires (comme celle de JavaCard), en particulier les nôtres. La section 3.3 dresse le paysage des outils et des techniques les plus utilisés au sein des méthodes formelles. Nous aborderons en particulier en sous-section 3.3.6 la description du système Coq qui a servi de support à la formalisation de JavaCard présentée au chapitre 4. La section 3.4 rapporte l'application des méthodes formelles à la vérification des plates-formes Java et JavaCard.

3.2 Les Critères Communs

Si la garantie absolue de la sécurité est impossible, certains organismes gouvernementaux — les premiers concernés, se sont intéressés de plus en plus près aux mesures possibles pour en assurer un niveau satisfaisant. Six pays (Allemagne, Canada, France, Grande Bretagne, Pays Bas et USA), disposant chacun d'une norme sur la sécurité des systèmes d'information, se sont ainsi concertés pour déterminer une approche commune pour l'évaluation de la sécurité des systèmes d'information. Cette concertation a abouti à la norme ISO-15408 dénommée par *Critères Communs* [55].

Garantie vs. Confiance Les Critères Communs souhaitent fournir un cadre à une réflexion plus organisée de la sécurité des systèmes d'information. L'objectif de la norme est de rationaliser les approches et la détermination des besoins de sécurité de tels systèmes. Plus précisément, les Critères Communs se proposent de fournir de solides bases à l'*attribution de confiance* aux systèmes se pliant à la norme.

Les Critères Communs comme mètre-étalon de la sécurité L'avantage indéniable d'une telle approche unifiée est de fournir un *espéranto* de la sécurité tant au niveau du concepteur du produit que de son utilisateur potentiel.

Un utilisateur peut recourir à une seule et même source, les Critères Communs, pour évaluer un produit et/ou un service en fonction de ses besoins de sécurité. Du point de vue du client, les Critères Communs sont la somme de toutes les précautions auparavant existantes.

Dans ce cas, les Critères Communs jouent un rôle important pour permettre aux utilisateurs d'identifier techniques et besoins de sécurité adaptés à leurs besoins organisationnels.

Ils servent de base à l'élaboration du référentiel nécessaire pour permettre l'évaluation de la solution proposée. Ce référentiel permettra à l'utilisateur de sélectionner parmi les fournisseurs ceux qui présentent la plus grande part des fonctionnalités attendues.

Les développeurs trouveront dans les Critères Communs un cahier des charges pour identifier les besoins élémentaires de sécurité et les appliquer à leur production.

Les écarts entre une offre produite et les attentes de fonctionnalités de la clientèle pourront ainsi être mieux analysés et permettront aux développeurs de mieux cerner les ajustements à effectuer pour que l'utilisateur puisse obtenir des solutions satisfaisantes.

Le cahier des charges induit par les Critères Communs est accepté internationalement, les concepteurs n'ont pas ainsi à procéder à une évaluation dans chaque pays où ils désirent diffuser leurs produits.

Le cycle de certification Critères Communs d'une application fait intervenir plusieurs acteurs, comme l'illustre la figure 3.1. Précisons le rôle de chacun :

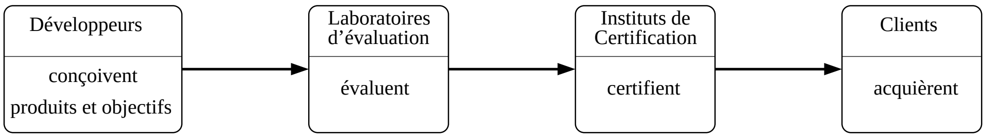

Fig. 3.1 – Processus de certification vu par les Critères Communs

– Les développeurs projettent l'implémentation du système. Le sous-ensemble du système traitant ou sensible aux considérations de sécurité est clairement défini. C'est la *cible de l'évaluation* (TOE pour *Target Of Evaluation*, dans la terminologie des Critères Communs). Les besoins en

terme de sécurité sont identifiés. Ceux-ci, selon les termes de la norme, impliquent la présence au sein du TOE d'un certain nombre de fonctionnalités de sécurité. Ce sont les *exigences fonctionnelles*. Les Critères Communs se prononcent aussi sur le cycle de production de l'architecture sécuritaire. Selon le degré de qualité visé, un certain nombre d'*exigences d'assurance* est requis. L'ensemble forme un document appelé *cible de sécurité*. À ce propos, les Critères Communs fournissent les *profils de protection* (dénommés PP dans la norme) afin de guider les concepteurs d'applications sécurisées à concevoir (de façon modulaire) l'ensemble des exigences (d'assurance et fonctionnelles) standards pour chaque famille d'applications ;
- Les laboratoires d'évaluation, en partenariat avec les concepteurs, analysent le bon respect des exigences, la correction du modèle, des mesures et des fonctionnalités de sécurité proposés.
  La norme se prononce aussi sur les procédés d'évaluation requis. L'analyse est ainsi menée par les évaluateurs en accord avec le *niveau d'assurance d'évaluation (EAL)* visé par les concepteurs ;
- Les instituts de certification, qui sont des organismes accrédités, analysent l'évaluation et, le cas échéant, valident la note EAL dénotant le niveau d'assurance du système évalué ;
- Les utilisateurs ont des exigences et un budget pour acquérir un produit disposant de fonctionnalités de sécurité bien précises. La certification Critères Communs de tout produit implique une présentation claire de ces fonctionnalités ainsi que l'évaluation acquise. Le client est en mesure de quantifier ses besoins, de comparer, d'ajuster en connaissance de cause son budget avec ces aspirations sécuritaires. L'acquisition est faite en toute connaissance de cause.

Critères Communs et méthodes formelles Selon les Critères Communs, la qualité d'une évaluation d'un TOE peut se mesurer en tenant compte de trois critères. Les trois dimensions de la qualité d'une évaluation sont :
- la portée de l'évaluation. La portée désigne la partie du TOE soumise à l'évaluation. On peut effectivement considérer une évaluation spécifique à chaque partie du TOE ;
- le grain de l'évaluation. Le grain d'une évaluation est le degré de détail avec lequel l'évaluation est menée ;
- la puissance de l'évaluation. La rigueur, la qualité et les fondements des outils et des méthodologies d'évaluation employés qualifient la puissance de l'évaluation.

La hiérarchie EAL des niveaux d'évaluation est claire à ce propos. C'est une échelle de 1 à 7, et les Critères Communs imposent l'utilisation des méthodes formelles pour atteindre les niveaux 5, 6 et 7. Selon la norme, les méthodes formelles fournissent les outils et les méthodologies proportionnant la plus forte puissance d'évaluation. La montée en puissance de 5 à 7 représente la montée en puissance des deux autres critères (portée et grain de l'évaluation) ainsi que le type de méthodes formelles utilisées. Par exemple, les outils de preuves sont requis pour atteindre le niveau 7.

Les Critères Communs fournissent ainsi un cadre d'application majeur aux méthodes formelles. C'est ce cadre qui a motivé l'origine de cette thèse. La vérification formelle de la plate-forme JavaCard que nous présentons a été pensé comme une contribution à sa certification EAL7.

3.3 Les Méthodes Formelles : Techniques et Outils

L'objectif de cette section est de fournir un panorama des différentes méthodologies et outils se réclamant des méthodes formelles. À défaut d'être exhaustif, nous aborderons les principales familles et insisterons plus particulièrement sur les outils ayant servi de support à la vérification formelle des plates-formes Java et JavaCard.

Les sujets comme "Que sont les méthodes formelles ?" ou encore "Quelles plus-values peut on attendre des méthodes formelles ?" ont déjà été largement débattus. Nous avons, à ce propos, déjà cité au début de ce chapitre quelques références contributives, ajoutons aussi [35, 46, 65, 136, 137] que nous utiliserons aussi comme guides.

Le problème central des méthodes formelles est donc de garantir un comportement donné d'un système informatique par une approche formelle. Au cœur des méthodes formelles se trouve la notion de *spécification*. Celle-ci, aussi appelée *cahier des charges*, est la modélisation du système cible et englobe, dans certains cas, la description du comportement désiré. Dans ce cadre, les méthodes formelles permettent de scinder le problème central en deux :

- Comment garantir, au niveau de la spécification, le comportement désiré ?
- Comment obtenir d'une spécification une implémentation conservant ce même comportement ? Ou alors, comment garantir qu'une implémentation jouit du même comportement qu'une spécification ?

L'existence de différentes méthodes formelles s'explique par la variété de méthodologies (et par conséquent d'outils) attaquant le problème sous cet angle : (1) *la validation d'un modèle* et (2) *la mise en relation d'une implémentation avec la spécification*.

Cette variété permet de considérer l'étude comportementale d'une spécification par animation, par transformation et par la preuve de propriétés. Mieux encore, elle fournit différentes modalités de liaisons formelles entre les spécifications et les implémentations. D'une part en permettant la dérivation d'implémentations à partir des spécifications, d'autre part en garantissant (par construction ou en permettant la preuve formelle) la correction de la synthèse effectuée.

Dans le reste de cette section nous allons nous attacher à discuter les différentes fonctionnalités énoncées.

3.3.1 Spécifier et analyser

La définition d'une spécification et l'analyse de son comportement peuvent être menées formellement, c'est-à-dire dans le cadre d'un formalisme mathématiquement bien défini. Les bénéfices que l'on en retire sont les suivants :

- la nature formelle du langage utilisé oblige une réflexion et une compréhension fine des mécanismes spécifiés ;
- la possibilité d'animer, ou même d'exécuter la spécification, et donc d'en d'observer directement le comportement, s'il existe une implémentation du formalisme mathématique sous-jacent. Une telle fonctionnalité permet le prototypage de système. L'avantage direct d'un prototype est qu'il est en général plus simple à mettre en œuvre que le système lui même et qu'il en permet ainsi la validation sans en impliquer l'implémentation ;
- au prototypage il faut ajouter que l'on reste dans un cadre formel. Celui-ci permet de mieux en appréhender le comportement : on dispose d'un contrôle "mathématique" du système modélisé. On peut donc raisonner et démontrer (à la main ou avec l'aide d'un ordinateur).

Les modèles formels vus comme des systèmes de transitions Le point communs aux différents langages de spécification est la notion modélisation par états et systèmes de transition. Les différents formalismes que nous allons maintenant aborder apportent chacun un support particulier à la notion d'état, de transition et de transformation d'état.

Les machines à états abstraits Proposées par Gurevich [82], les *machines à états abstraits* (ASM, pour *Abstract State Machines*, mais aussi appelées *algèbres évolutives*) forment un langage de spécification dans lequel les notions d'*état* et de *transformation d'états* sont centrales. Un système y est décrit par la définition d'états et par un ensemble fini de règles de transition. Cet ensemble décrit sous quelles conditions (appelées gardes) un ensemble de transformations (modifications de l'état interne de la machine) ont lieu permettant la transition d'un état du système à un autre. Ces transitions ne sont pas nécessairement séquentielles. Le formalisme permet de prendre en compte la configuration où pour un état de la machine plusieurs transitions sont éligibles. Le formalisme ASM, puisqu'il se révèle aussi puissant que les machines de Turing, peut être utilisé comme langage de spécification exécutable. La notion d'exécution d'un ASM est la notion classique d'exécution d'un système de transition.

Citons ASM_Gopher[146] qui est une implémentation de ce formalisme, et qui a servi de cadre à une formalisation de Java. Citons aussi B [1] que nous retrouverons dans les systèmes à formalisme ensembliste.

La modélisation par automates Un autre type de systèmes de transitions utilisé pour la spécification sont les *automates* [3, 4, 100, 110, 134, 160, 164]. Dans ce cadre, c'est le comportement concurrent du système spécifié qui est au cœur de la modélisation. L'idée centrale est de définir comment réagit le système face à un ensemble de stimuli. Ce formalisme est ainsi particulièrement adéquat à la spécification de systèmes réactifs, concurrents, communicants ou encore de protocoles. Il l'est cependant moins pour des systèmes où l'ensemble des états et l'ensemble des transitions sont difficiles à exprimer.

La théorie des ensembles et la théorie des catégories Nous regroupons ces deux théories mathématiques car elles offrent, au niveau des spécifications, une expressivité semblable. Les états, dans ces deux formalismes, sont décrits en terme de structures mathématiques expressives comme des ensembles, des relations ou des fonctions. Les transitions sont exprimées en termes d'*invariants* et de *pré* ou *post-conditions*.

Z [145], VDM[94] sont des exemples classiques de méthodes formelles dont le langage de spécification repose sur la théorie des ensembles. Ils ont été à l'origine de beaucoup d'autres systèmes. RAISE [70], par exemple, est une évolution de VDM. la méthode B [1] est aussi un système de spécification héritier des traditions Z et VDM. En plus de son langage ensembliste, la méthode B se base sur la notion de machines à états abstraits. Les systèmes SPECWARE [95] et CHARITY [47] offrent quant à eux un formalisme fondé sur la théorie des catégories.

Spécifications algébriques Une approche classique à la spécification est la spécification par algèbres multi-sortées. Une algèbre multi-sortée est une collection d'ensemble de données (un ensemble pour chaque type de données), une collection de fonctions sur ces ensembles correspondant aux fonctions du programme modélisé ainsi qu'un ensemble d'axiomes spécifiant les propriétés de base des fonctions de l'algèbre. Un tel formalisme permet de s'abstraire des algorithmes utilisés pour encoder le calcul voulu en se concentrant sur la représentation des données et sur le comportement des fonctions par rapport aux entrées et aux sorties. Une spécification algébrique se présente sous la forme d'une suite de déclaration de sortes, de signatures de fonctions et d'axiomes déclarant le comportement de base de chaque symbole de fonction. OBJ [68], CLEAR [37], Larch [69] et ACT-ONE [63] sont des méthodes formelles à langage de spécification algébrique.

La modélisation déclarative Une classe importante de langages de spécification est la classe regroupant les langages fonctionnels, les langages logiques, les langages basés sur la réécriture et, enfin, les langages pour la définition de sémantiques formelles. Ceux-ci reposent sur des bases mathématiques bien connues.

Les formalismes logiques, comme Prolog [148], proposent un langage dans lequel la modélisation est centrée sur la notion de prédicat. Les données sont représentées à l'aide de types de données simples mais suffisamment expressifs (comme les listes), une opération est décrite par ses propriétés comportementales, à l'image des axiomes des spécifications algébriques. Prolog est un langage de programmation, et permet donc d'exécuter les spécifications.

Les langages fonctionnels proposent un cadre de spécification dans lequel la notion de fonction est centrale. À la base, nous trouvons le λ-calcul [14, 15] qui est le fondement de tous ces formalismes. Le λ-calcul a la même expressivité que les machines de Turing et propose de formuler toute opération, et même toute donnée, en terme de fonctions d'ordre supérieur. Les langages comme Scheme [61], SML [115], Haskell [157] ou OCaml [108], et les systèmes de preuve comme ACL2 [96], DECLARE [154], Coq [50], PVS [142], HOL [79] ou encore ISABELLE [126] proposent des extensions au λ-calcul. Ces extensions sont plus aisément utilisables que le calcul original et proposent de nombreux types de données de base, de puissants opérateurs de construction de types (les types inductifs en sont un exemple) et de puissants mécanismes de définition de fonctions. L'exécution dans ces langages repose sur la notion de réduction qui s'assimile à la notion de calcul en mathématique, c'est-à-dire de l'application des arguments à une fonction. Les langages de ces systèmes sont au moins des langages d'ordre supérieur (certains systèmes, comme Coq, proposent des extensions encore plus expressives) et offrent ainsi une flexibilité et une puissance d'expression appréciable. Soulignons, pour finir, que les langages des assistants de preuves basé sur la théorie des types, par le biais de l'isomorphisme de Curry-Howard [15, 144], peuvent être utilisés comme langages logiques d'ordre supérieur.

Les systèmes de réécriture [9, 30, 98] comme ELAN [32] ou SPIKE [34] proposent des langages très proches de ceux des spécifications algébriques, à la différence que les axiomes sont remplacés par des équations déterminant le comportement calculatoire des symboles de fonction. À l'image du λ-calcul, l'exécution repose sur la notion de réduction. Dans le cadre de la réécriture, la relation de réduction est défini par les équations.

Pour finir avec les formalismes déclaratifs, citons les langages qui se destinent à l'expression de sémantiques formelles. Citons Action Semantics (AS) [119, 120], ASF+SDF [89, 159], Centaur [33], RML [127] et Letos [85]. Tous ces systèmes offrent :

- un environnement d'expression de sémantiques, en particulier de sémantiques de langages de programmation. Ces langages sont proches des langages algébriques dans lequel les axiomes sont remplacés par les règles sémantiques régissant le comportement sémantique des symboles définis ;
- un support d'exécution des sémantiques définies.

3.3.2 Spécifier et démontrer

Pour garantir un comportement donné d'une spécification, l'animation ou l'exécution ne sont évidemment pas suffisantes. Une démonstration rigoureuse est impérative. On parle à ce propos de *vérification formelle* et non plus seulement de spécification ou de modélisation. Plusieurs procédés sont possibles. J. Rushby [136] propose un classement à trois niveaux des méthodes de démonstrations.

Le premier niveau regroupe les outils fournissant un cadre formel à la démonstration sans pour autant en fournir un support. Les démonstrations y sont faites “à la main”. Une démonstration est validée lorsqu'elle convainc les évaluateurs.

Les outils du deuxième niveau proposent en plus un système formel permettant une formulation plus rigoureuse de la démonstration (Par exemple, l'usage de la langue naturelle, possible au niveau précédent, disparaît). Celle-ci est cependant toujours menée manuellement.

Le troisième niveau regroupent les outils intégrant la formulation d'une démonstration dans leur sein même. Des trois niveaux, celui-ci propose le plus haut niveau de rigueur et de garantie. L'avantage indéniable est que la preuve est ainsi *exprimée* et *vérifiée* dans un cadre qui ne laisse pas de place à l'ambiguïté, comme les problèmes d'interprétation. Les problèmes de validité sont de plus factorisés, la validité de toute démonstration repose sur celle de l'outil. Nous abordons ici les outils appartenant à ce niveau. Ceux-ci permettent d'*exprimer* un modèle et de *raisonner formellement* et *mécaniquement* sur celui-ci.

Les systèmes de preuve

La construction de démonstrations dans les systèmes de preuve se base sur un modèle déductif induit par le système formel propre à chacun d'eux. Deux facteurs antagoniques ont un impact important sur le comportement déductif des moteurs de preuve. D'une part l'expressivité logique permet l'énoncé de propriétés et de déductions complexes, d'autre part la simplicité du formalisme logique permet une grande automatisation de la déduction. Les différents systèmes de preuve que nous allons présenter offrent chacun un compromis particulier entre expressivité et automatisation.

Les démonstrateurs de théorèmes préfèrent privilégier l'automatisation de la déduction au pouvoir d'expression. La construction de la preuve, une fois le moteur de preuve paramétré, est automatique. C'est le cas de la logique des clauses de Horn propre à Prolog, des systèmes de réécriture du premier ordre de SPIKE et de ELAN ou encore du fragment de la logique du premier ordre de ACL2. Parce que nous n'utiliserons pas directement ces outils nous ne rentrerons pas les détails de la réécriture, ou des clauses de Horn. Soulignons cependant qu'à la différence des outils de vérification de modèles, les démonstrateurs de théorème peuvent raisonner sur des ensembles infinis. Les techniques de raisonnement par induction sont un exemple de techniques permettant le raisonnement sur de tels ensembles.

Les assistants de preuve Par opposition aux démonstrateurs de théorèmes, le choix des assistants de preuve est de privilégier l'expressivité de la logique véhicule de la preuve. Dans ce cas, les logiques en question, les logiques d'ordre supérieur, sont indécidables. Il n'existe donc pas de procédures de décision construisant une preuve pour une propriété arbitraire. Si ce fait parait restrictif, il faut cependant souligner que beaucoup de propriétés ont besoin de la puissance et de l'élégance de telles logiques pour être convenablement exprimées et démontrées. Le procédé de démonstration demeure déductif, mais l'aide de l'utilisateur est nécessaire pour conclure la preuve. De tels systèmes sont aussi appelés *vérifieurs de théorèmes* car ils fournissent les moyens à l'utilisateur de construire une démonstration. Mais, puisque que l'utilisateur n'est pas à l'abri d'erreurs, ces systèmes doivent *vérifier* la validité de la preuve construite.

Deux approches à l'expressivité cohabitent dans le domaine des assistants de preuve. La première consiste à donner la possibilité à l'utilisateur de définir la logique dans laquelle il désire exprimer ses preuves. Cette logique est alors dénommée *logique objet*. C'est cette *approche axiomatique* qu'a adopté le système

Isabelle. L'autre approche consiste à fournir un langage de base suffisamment expressif pour pouvoir formuler une grande partie des mathématiques. On retrouve cette *approche intégrée* dans les systèmes comme Coq, PVS, DECLARE ou encore HOL.

Dans certains systèmes, comme Coq, les démonstrations sont des objets à part entière du langage, au même niveau que les énoncés et la spécification. Nous en verrons les avantages à la section 3.3.3. Dans ces systèmes, la démonstration peut être exprimée directement par l'utilisateur. Cependant, une démonstration n'est pas en général facile à exprimer. Ainsi, le procédé standard que l'on retrouve dans tous les assistants de preuve est la construction interactive de la preuve par l'application de fonctions de haut niveau de manipulation de preuve. Une telle fonction, appelée *tactique*, code un pas de preuve. L'état de preuve est habituellement représenté par une pile où chaque élément est un séquent (i.e. une pile d'hypothèses et une conclusion à prouver). Établir la preuve d'une propriété P consiste à appliquer dans un certain ordre un ensemble de tactiques de façon à construire un arbre de preuve liant axiomes et théorèmes à la conclusion P. Dans sa forme la plus simple, une tactique exprime un pas de preuve de base (comme le *modus ponens*). Cependant les tactiques ne se limitent pas au raisonnement "atomique". Une tactique peut permettre un raisonnement complexe ou même une preuve complète comme c'est le cas avec les tactiques implémentant des procédures de décision sur un fragment décidable de la logique d'ordre supérieur.

Pour finir, signalons le support à la preuve du système B [1], qui, s'il ne repose pas sur la logique d'ordre supérieur — il s'agit d'un formalisme logique du premier ordre, permet la preuve interactive des lemmes qui n'ont pu être prouvés automatiquement.

La vérification de modèles

La vérification de modèles [45] (*model checking* en anglais) est une technique de vérification de systèmes (concurrents) à état finis (comme les automates, voir section 3.3.1). Dans de tels systèmes, les propriétés attendues du modèle sont exprimées par des formules d'une logique temporelle, et des algorithmes symboliques efficaces sont utilisés pour traverser le modèle dans sa totalité afin de vérifier si toutes les configurations (ou états) possibles valident les propriétés. L'ensemble de tous les états s'appelle *l'espace d'états*. Lorsque le système considéré possède un espace d'état fini, les algorithmes de vérification de modèles peuvent être utilisés pour obtenir une preuve automatique. Dans le cas ou la propriété n'est pas vérifiée un contre-exemple est exhibé.

Malheureusement, un inconvénient sérieux à cette approche est le problème de l'explosion de l'espace d'états. En effet le graphe des transitions grandit exponentiellement pendant que la taille du système croît linéairement. La conséquence immédiate est que rapidement, pour aussi efficace que les algorithmes de vérification soient, l'exploration de l'espace considéré devient impraticable.

Plusieurs techniques ont été présentées pour faire face à ce problème. Citons seulement une technique que nous retrouverons aux chapitres 4 et 5, l'*abstraction*. L'idée est de proposer un modèle simplifié, un *modèle abstrait* (et dont l'espace d'état peut être exploré) respectant certaines propriétés garantissant que si la propriété visée est valide pour le modèle abstrait, alors elle l'est aussi dans le modèle original.

Annotation de programmes et logique de Hoare

Pour finir avec les techniques de preuve, abordons les outils permettant l'annotation de programmes. Une annotation de programme est un énoncé placé au cœur même du code du programme dont on cherche à vérifier le comportement. L'annotation d'une méthode, d'une fonction ou d'un bout de code est censé indiquer les conditions préalables à l'exécution de la portion de code concernée ainsi que l'état "logique" dans lequel se trouve le programme à la fin de l'exécution de celle-ci. Le formalisme logique utilisé est appelé une *logique de Hoare*. Ces logiques prennent une forme très variée selon le langage cible. En effet si la syntaxe des annotations est assez standard, la sémantique (*axiomatique*) des annotations est totalement liée au langage de programmation. Dans le cadre de Java, plusieurs systèmes d'annotations existent comme JML [93], Esc/Java [64]. Des outils comme le LOOP Tool [26, 130], Bandera [12, 51, 60], Jive [113] ou encore Krakatoa [99] permettent de vérifier la correspondance du code avec les annotations en associant des techniques de vérification de modèles avec des techniques de preuve assistée (pour ce qui est du Loop Tool) ou en associant des techniques de vérification de modèles avec des mécanismes d'abstraction (Bandera). L'avantage indéniable d'une telle approche est que c'est l'implémentation (en langage source) qui sert de base à la vérification, et non pas une spécification. En fait un modèle est construit par l'outil à partir des annotations et du programme. De fait, cette approche propose une alternative satisfaisante au problème central des méthodes formelles.

3.3.3 Spécifier et dériver

Jusqu'à maintenant nous avons abordé les techniques et les outils permettant d'appréhender la première partie du problème central posé aux méthodes formelles. Reste maintenant la deuxième partie : Étant donné une spécification jouissant des propriétés désirées, comment obtenir une implémentation dont le comportement est celui de la spécification ? Nous verrons que plusieurs solutions existent. Elles sont de deux ordres. D'une part la spécification est le programme lui même — C.Q.F.D., d'autre part une implémentation est produite, par des procédés que nous décrirons par la suite, à partir de la spécification. Rappelons l'approche par annotation de programme qui aborde le problème sous un angle bien différent en travaillant directement sur le programme.

Surgit, dans le cas de la dérivation d'implémentations, le problème de la correction de la synthèse. Deux approches cohabitent. La première, du fait des bonnes propriétés de base des mécanismes de dérivation, restreint les possibilités de synthèses et ne permet que la production de code vérifiant les propriétés de la spécification originale. C'est ce qu'on appelle le raffinement correct par *construction*. La deuxième génère des obligations de preuve permettant de garantir (sous l'hypothèse que celles-ci sont vraies) que la dérivation produit bien une implémentation correcte aux vues des propriétés désirées.

La correction peut alors être validée soit manuellement soit, de préférence, par machine (en utilisant les outils de vérification formelle “de niveau 3”). Si la démonstration “de niveau 3” est envisageable dans toute sa généralité, (c'est-à-dire que le procédé de synthèse est lui même formalisé et validé) nous obtenons alors un procédé universellement validé dont on peut affirmer dorénavant qu'il est correct par *construction*. Sans cela toute synthèse doit être validée individuellement. Cette petite digression nous rappelle simplement que la vérification formelle est, de fait, omniprésente à tout stade du problème central.

Raffinement

Le raffinement est une technique permettant la synthèse pas à pas d'un programme à partir d'une spécification. L'idée est de donner un niveau de précision de plus en plus important à la spécification. Chaque degré de précision supplémentaire représente un choix d'implémentation, comme par exemple le choix de l'algorithme implémentant une fonction, ou encore le choix de la représentation concrète d'un type de donnée (passer par exemple d'une représentation ensembliste à une liste chaînée). Chaque pas doit être prouvé correct pour que le programme final jouisse des propriétés de la spécification originale, la bonne nouvelle étant que la preuve individuelle de correction de chaque pas est bien plus facile à mettre en œuvre que celle de la correction globale. C'est cette approche qu'ont adoptée les systèmes de spécification Z, VDM, B et SPECWARE. Soulignons que dans le cas de SPECWARE, les bonnes propriétés du formalisme sous-jacent (la théorie des catégories) permettent une correction du processus de raffinement par construction.

Extraction

Le calcul des constructions inductives (CCI), qui est au cœur de Coq, est une extension typée du λ-calcul. Par l'isomorphisme de Curry-Howard [15, 144], ce calcul est aussi une logique d'ordre supérieur minimale *constructive*. La conséquence est que tout énoncé logique est aussi une spécification et que toute preuve qui s'exprime en CCI est un programme qui respecte une spécification (l'énoncé du théorème). Ce paradigme fort permet de plus l'*extraction* du contenu calculatoire d'une preuve (le programme qu'elle contient) vers un langage de programmation.

Soit T le théorème suivant :

$$\forall x, y \in \mathbb{N}. \exists q, r \in \mathbb{N}. (y = (q \times x + r) \wedge 0 \leq r < x)$$

Une preuve de T est, dans ce cadre, une fonction qui prenant deux entiers x et y calcule la paire (q, r) témoignant de la validité de $(y = (q \times x + r) \wedge 0 \leq r < x)$. Cette paire est bien entendu la paire regroupant quotient entier et reste de la division de y par x.

De façon générale la preuve d'un théorème de la forme $\forall x \exists y.(R\ x\ y)$, si elle existe, est une fonction qui pour tout x calcule le y tel que $(R\ x\ y)$ soit vrai.

Coq propose une extraction vers les langages fonctionnels typés Haskell et OCaml. Remarquons que, dans ce cadre, l'extraction du programme correspondant à la preuve t d'une propriété/spécification T se fait automatiquement en une seule étape.

Exécution

Spécifier à l'aide de langages de programmation est un moyen *de facto* de disposer d'implémentations. Les langage logiques comme Prolog et les langages fonctionnels (Scheme, ACL2, Haskell, SML, OCaml que l'on a déjà cités) en sont des exemples. Le problème simple qui se pose à cette approche est la pertinence du langage en terme de retombées industrielles. Prolog constitue-t-il vraiment un langage d'implémentation viable pour l'industrie ?

3.3.4 Spécifier et transformer

Nous avons souligné à propos de la vérification de modèles qu'il était souhaitable dans certain cas de transformer une spécification de façon à ce qu'elle puisse être manipulée par les méthodes de vérification.

Ce constat se transpose hors du simple cadre de la vérification de modèles. Il est souvent utile, et même nécessaire, de construire des variations de la spécification afin de cacher des détails obscurcissant la vérification ou au contraire de rajouter un niveau de détails afin de prendre en compte de nouveaux comportements. De façon plus générale, la possibilité de construire des variations au modèle initial permet d'envisager la vérification formelle d'un comportement de façon *modulaire*. Un comportement peut être décomposé en *facettes*. Chaque facette de ce comportement ne concerne qu'une partie bien définie de la modélisation globale. Les variations permettent de mettre cette partie en exergue par rapport au reste du modèle. Modulo preuve de compositionnalité des facettes et mise œuvre des variations, l'ensemble de l'étude des facettes est plus simple à mettre en œuvre que celui du comportement global.

L'outil JaKarTa que nous présentons au chapitre 5 et ASF+SDF supportent la construction systématisée de telles variations.

3.3.5 Combiner le meilleur des mondes

Clarke et al. [46] constatent qu'aucun outil semble solutionner de manière complètement satisfaisante le problème central des méthodes formelles. Si les assistants de preuve proposent des outils solides pour la vérification formelle, ils sont cependant assez difficiles à utiliser et manquent d'automatisation. Si certains outils de spécification proposent un traitement vertical (des spécifications aux programmes) complet, ils leur manquent des fonctionnalités de preuves. La plupart des outils ne fournissent pas de support particulier aux mécanismes de transformation de spécification.

Si des efforts ont été cependant mené pour agrémenter fonctionnellement et individuellement certains outils (citons à ce propos B qui propose aussi bien une solution verticale à la spécification que des mécanismes de preuve, citons aussi

PVS et Coq[161] qui ont proposés des modules de vérification de modèles plus ou moins aboutis), le résultat n'est pas entièrement satisfaisant.

Le constat évident est qu'à défaut de pouvoir disposer d'un outil "tout en un", la combinaison de systèmes est une alternative intéressante.

Des systèmes comme PROSPER [57], KEY [2], ASF+SDF ou encore JaKarTa sont des exemples de cet effort de combinaison, de multiples compétences. PROSPER propose une architecture de communication centrée autour du système de preuve HOL permettant aux système de spécification, entre autres, de dialoguer et de déléguer les tâches de preuve. Sur le même principe, ASF+SDF construit un module de communication (appelé ToolBus) entre les nombreux composants le constituant. Si ceux-ci ont plutôt l'objectif de fournir un cadre à la conception de langages de programmation plus que la vérification formelle, ASF+SDF regroupe des fonctionnalités comme la spécification de sémantiques par un langage algébrique basé sur la réécriture conditionnelle, un éditeur graphique, un interpréteur conjointement avec un module de transformation de spécification (de sémantique). Le système KEY, quant à lui, propose une intégration de modules de preuve d'annotations de programme au sein un système d'aide à la conception.

Dans ce contexte, JaKarTa se propose de faciliter la spécification et la preuve en combinant mécanismes de transformation de spécifications (abstraction et raffinement), outils de spécification et outils de preuve. JaKarTa propose :

- une architecture centrée sur une plate-forme de spécification ;
- des mécanismes de communication entre différents outils et la plate-forme de spécification. Actuellement les systèmes connectés intègrent langages de programmation et systèmes de preuves (démonstrateurs de théorèmes, et assistants de preuve) ;
- des mécanismes d'administration et de mécanisation des démonstrations sur les spécifications JaKarTa dans l'assistant de preuve Coq ;
- un cadre adéquat à la transformation de spécifications JaKarTa, ainsi que l'automatisation des démonstrations de correction de celle-ci. Notons que ces transformations jouent ici un rôle proche du raffinement dans les systèmes de spécification. Ce que propose JaKarTa c'est de permettre la définition de variation de spécification, d'en automatiser la synthèse, de générer les obligations de preuve résultantes et d'en automatiser la vérification. Nous soulignerons section 3.5 l'importante d'un tel support dans le cadre de formalisations larges.

Soulignons un aspect essentiel de JaKarTa. Il s'agit d'une combinaison d'outils qui, contrairement aux autres systèmes présentés, a été pensé et déployé pour répondre à un besoin bien précis, celui de proposer un support adéquat à vérification formelle de plates-formes logicielles comme JavaCard.

3.3.6 Un rapide survol du système Coq

Nous avons utilisé intensivement dans nos travaux l'assistant de preuve Coq. Nous détaillerons donc dans cette section les éléments nécessaires à la compréhension du code Coq que nous présenterons au chapitre 4. Une présentation complète de cet outil est exposée dans [29, 50, 73].

Le système Coq est un assistant de preuve généraliste pour la spécification et la démonstration en mathématique (au sens large du terme). Il est basé sur le calcul des constructions inductives et, à travers à l'isomorphisme de Curry-Howard, intègre un langage de spécification riche et une logique des prédicats d'ordre supérieur.

Le langage de spécification est à bien des égards plus puissant que les langages de programmation standard. Par exemple, les structures complexes sont définissables en tant que types enregistrement *dépendants* ; de même, les types de données complexes sont définissables en tant que types *inductifs*.

De plus, le langage de spécifications permet les définitions de fonction par (co-)récursion sur les types (co-)inductifs. De façon à avoir une vérification de type décidable, Coq exige que toute fonction (récursive) définie termine. Pour cette raison, les fonctions récursives Coq sont définies par récursion structurelle. La récursion bien fondée est cependant possible via une définition structurelle sur la preuve de bonne fondation de l'appel récursif (consulter [10, 73] pour plus de précision).

Continuons notre présentation avec quelques notations. Le symbole `*` dénote le produit cartésien de deux types, `(a,b)` une paire, `[x:A] b` une λ-abstraction, `(x:A) B` l'espace dépendant de fonctions de `A` dans `B` et `A→B` un espace non dépendant de fonctions de `A` dans `B`. Un type inductif est déclaré par le mot clé **`Inductive`** conjointement avec son nom, avec éventuellement des paramètres, son type et la liste de déclaration de ses constructeurs (nom et signature). Voici deux exemples classiques, le type des entiers naturels et le type des listes polymorphes :

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

Inductive list [A : Set]  : Set :=
      nil : (list A) | cons : A → (list A) → (list A).
```

Un enregistrement `R` est déclaré avec le mot clé **`Record`** suivi de son nom, de son type, et d'une description (nom et type) de ses champs. Il est internement représenté dans Coq comme un type inductif à un seul constructeur. Les sélecteurs sont des fonctions (définies par analyse de cas) ainsi nous écrivons `(l a)` au lieu du plus standard `a.l`. Donnons en exemple l'ensemble des rationnels `Rationnel`. Un rationnel est défini en faisant appel au constructeur `Creer_rationnel` et dont les champs sont son signe, le numérateur le dénominateur, la preuve que le dénominateur n'est pas 0 (où les symboles `/\` représentent la conjonction) et la preuve que la fraction est sous sa forme irréductible :

```
Record Rationnel : Set := Creer_rationnel {
  signe           : bool;
  numerateur      : nat;
  denominateur    : nat;
  Rat_bottom_cond : ¬ O=denominateur;
  Rat_irred_cond  :(x,y,z:nat)(mult x y)=numerateur
                          /\(mult x z)=denominateur → x=(S O)
}.
```

Pour finir, les définitions sont introduites par le mot clé **Definition** et la concordance de forme sur un type inductif est introduite par la notation **Cases** comme montre l'exemple intuitif suivant :

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

Comme de coutume en théorie des types dépendants, le langage de spécification oblige à ce que toute fonction soit totale. Néanmoins, des fonctions partielles peuvent être représentées dans Coq en utilisant trois techniques :
- en fixant une valeur arbitraire de retour pour toute application de la fonction hors de son domaine ;
- en paramétrant la fonction avec la preuve que l'argument passé est dans son domaine de définition ;
- en utilisant la monade *exception* qui est introduite par la définition inductive suivante :

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

Par exemple, la fonction qui calcule la tête d'une liste en utilisant le type `Option` est définie par :

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

Dans ce cas, lorsque la liste est vide une erreur, le constructeur `None`, est retournée.

Les fonctions récursives sont introduites par le mot clé **Fixpoint**. La fonction `dernier_element` est un exemple :

```
Fixpoint dernier_element [A:Set,l:(list A)] : (Option A) :=
  Cases l of
    nil          ⇒ (None A) |
    (cons x nil) ⇒ (Some A x) |
    (cons x tl)  ⇒ (dernier_element tl)
  end.
```

Nous utiliserons aussi quelques mécanismes propres à Coq qui sont des extensions au langage de base :

1. le mécanisme de *section*, qui permet de spécifier des définitions paramétrées de façon plus claire et aisée. Par exemple, toutes les définitions de base sur des listes appartiennent à une section de la forme :

   ```
   Section Poly_List.
   Variable A: Set.
   ...
   End Poly_List.
   ```

   Hors de la section, les définitions dépendront du type `A`, qui représente le type des éléments de la liste ;

2. les *arguments implicites*, qui soulagent les utilisateurs du fardeau de rendre toutes les annotations de type explicites. Les arguments implicites rendent de plus les spécifications plus lisibles.

   Ce mécanisme permet d'oublier certains arguments pendant une application de fonction. Par exemple, les arguments implicites permettent aux utilisateurs d'écrire `(cons (S n) l)` au lieu de `(cons nat (S n) l)`, puisque le type des éléments de la liste peut être déduit de `S n` ;
3. le mécanisme de *Coercions*, qui intuitivement généralise le sous-typage et qui est employé pour implicitement traduire des valeurs d'un type dans un autre. Par exemple, après avoir déclaré une coercion des entiers naturels vers les entiers relatifs, il est possible d'appliquer n'importe quelle fonction sur les entiers relatifs à des entiers naturels. La syntaxe de Coq pour déclarer une telle coercion est :

   ```
   Coercion nat_int : nat ↣ Z.
   ```

   Où il est supposé que `nat_int` soit un identificateur auparavant défini avec le type `nat` $\rightarrow$`Z` ;
4. les *noms qualifiés* permettent aux constructions Coq (définitions, lemmes, théorèmes...) d'être définies avec le même nom mais dans des modules (ou bibliothèques) différents. Si une construction `c` est définie dans un module `m`, l'utilisateur peut lui faire référence par la notation `m.c`.

3.4 Modélisation et vérification formelle de Java et de JavaCard

Nous allons, dans cette section présenter un état de l'art de la modélisation et de la vérification formelle de Java et de JavaCard.

Nous nous restreignons dans cette section à la présentation des travaux dont l'objectif est la vérification formelle (supportée par un outil) des plates-formes Java et JavaCard tant au niveau de la plate-forme elle-même qu'au niveau des applications pouvant y être exécutées. Cette restriction correspond aux domaines de contribution des travaux présentés chapitres 4, 5 et 6. Notre contribution directe à la vérification formelle de la plate-forme JavaCard se situe plus particulièrement sur la modélisation et la validation de différentes machines virtuelles, c'est pourquoi nous insisterons plus particulièrement sur les travaux de ce ressort. Pour une approche plus générale se reporter à l'article de L. Moreau et de P. Hartel [86].

3.4.1 Vérification de la plate-forme Java/JavaCard

Defensive Virtual Machine R. Cohen [49] a été le premier, à notre connaissance, à proposer une formalisation d'un sous-ensemble convaincant de la JVM sur un outil de démonstration de théorèmes (le système ACL2, qui, rappelons le, propose une logique du premier ordre). L'objectif principal de ses travaux a été de fournir une sémantique opérationnelle *exécutable* de la JVM (appelé dJVM) implémentant complètement la spécification SUN (dans le sens où les clauses *must* ou *must not* des spécifications SUN sont assurées par l'implémentation). Cet effort conduit à la définition d'une machine défensive. Quant à la portée de la formalisation, la JVM n'a pas été formalisée dans toute sa totalité. Un peu moins de la moitié des instructions manque, et certaines fonctionnalités aussi. Citons, parmi d'autres, le multithreading, le chargement dynamique de classe, le ramasse miettes, les interfaces, la récupération d'exception. Si aucune preuve n'a été développée sur la dJVM, celle-ci est exécutable et par conséquent permet le test. Elle constitue néanmoins une plate-forme possible pour le raisonnement.

J. Strother Moore, dans [118, 117], étend la dJVM en implémentant la quasi totalité des instructions et en apportant certaines caractéristiques comme le multithreading, et développe une vérification formelle de la sûreté du typage dans un cadre du premier ordre (ACL2).

Project Bali L'une des modélisations les plus accomplies a été menée par le projet Bali [11]. Formalisée en Isabelle/HOL, celle-ci comprend une grande partie de la plate-forme Java. C'est-à-dire :

- un système de type et une sémantique opérationnelle des deux variantes du langage (langage source et bytecode)[125, 162, 131], avec la démonstration de la sûreté du typage ;
- un compilateur [149], un vérificateur de bytecode abstrait [97, 124] et un vérificateur de bytecode léger accompagnés d'une preuve de leur correction ;
- une sémantique axiomatique correcte et complète permettant la vérification de programmes Java.

Ces travaux sont aboutis dans le sens où ils prennent en compte tous les composants de la plate-forme Java, du langage source à la machine virtuelle. Ajoutons que leur formalisation prend en compte les problèmes liés aux sous-routines et ceux liés à l'initialisation des objets bien que leur méthodologie, à ce propos, se distingue de la spécification SUN. Notons aussi qu'aucune adaptation explicite à JavaCard n'a été menée. Les travaux de C. Pusch [131], qui s'inscrivent aussi dans ce projet et qui se sont basés sur les formalisations papier de Qian [132], présentent une des premières formalisations de niveau 3 (voir classification de Rushby section 3.3.2) de la JVM.

JBook Börger, Schmid et Stärk proposent, dans leur livre [146], une modélisation de l'architecture Java basée sur les ASM et écrite avec ASM_Gopher. Ceci inclut la définition modulaire de la sémantique du langage Java, la formalisation du compilateur, de plusieurs machines virtuelles et enfin du BCV. Leur travaux présentent une méthodologie de développement hiérarchique basée sur le raffinement.

Le premier niveau de leur sémantique du langage source couvre les aspects impératifs et séquentiels, puis viennent s'ajouter les extensions couvrant progressivement tous les aspects propres à Java : la modularité, l'orientation objet, le traitement des exceptions et la concurrence.

Quant aux machines virtuelles, la formalisation commence par un modèle d'une machine *diligente* — c'est-à-dire une machine offensive, qui assume que tout contrôle de bon typage est positif, associée à une machine de vérification de contraintes de typage. Le processus de formalisation est similaire à la formalisation de Java : la machine virtuelle est raffinée de telle façon que les différentes fonctionnalités soient intégrées de façon modulaire. Sur cette base, une machine défensive qui vérifie explicitement le bon typage avant toute exécution est construite. Notons (1) que la machine virtuelle formalisée considère le chargement dynamique et la phase de liaison (2) mais que celle-ci ne considère pas de ramasse miettes (3) et que certaines hypothèses sur la forme d'un programme compilé permettent d'alléger et donc de prendre en compte le problème des sous-routines par le BCV.

La phase de compilation est ensuite formalisée comme une fonction prenant la description d'un programme et le transformant dans une représentation acceptable par les machines virtuelles et respectant certaines contraintes (de bonne formation, entre autres).

Le principal résultat de ce projet est la démonstration (papier) de la correction de l'ensemble de la formalisation : un programme respectant certaines conditions bien définies comme la bonne formation, passant avec succès compilation et vérification sera exécuté sans violation des vérifications réalisées par la machine défensive.

Bien que leurs travaux soulignent l'étroite relation entre les machines offensive et défensive, leur approche semble moins propice à l'automatisation de la construction et de la validation des machines virtuelles. Par exemple, la construction d'un outil permettant la synthèse systématisée d'une machine virtuelle défensive à partir de sa contrepartie offensive ne semble pas évidente.

GEMPLUS Dans une série d'articles, L. Casset, J.-L. Lanet et A. Requet [41, 102, 135] rapportent l'utilisation de la méthode B [1] pour la spécification de la machine virtuelle JavaCard. La méthodologie employée suit celle que nous décrivons au chapitre 4, c'est-à-dire qu'une machine virtuelle offensive est construite à partir d'une machine virtuelle défensive. Cependant le procédé effectif diffère. Bien que la première phase soit aussi constituée par la construction de la machine défensive, ils construisent la machine offensive et le vérifieur de bytecode par raffinements successifs. La preuve de correction de la machine offensive est exprimée en termes de correction de la phase de raffinement. Les parallèles entre leurs travaux et les nôtres sont nombreux, néanmoins cette formalisation ne considère pas la machine virtuelle dans sa totalité, mais il s'agit ici plus d'un problème de granularité que de portée. Soulignons aussi que l'automatisation des preuves propre au système B est un facteur décisif dans le processus de validation entamé. Cependant la méthode B ne dispose pas de l'expressivité nécessaire à l'établissement de propriétés complexes comme la terminaison ou l'analyse de flux de donnée pourtant nécessaire à la vérification du bytecode.

Dans un autre registre, L. Casset [40, 42] propose le développement et la validation formelle (toujours en B) d'un vérifieur de bytecode. C'est, comme le BCV de Trusted Logic (décrit dans cette section), un vérifieur pensé pour être embarqué sur la carte. Les optimisations apportées qui permettent de l'embarquer sur la carte à puce prennent origine dans le paradigme du "*proof carrying* *code*" [122]. L'idée est d'ajouter au bytecode un minimum d'information de type en permettant une vérification efficace. Par raffinement, un BCV en C a été obtenu et a été placé avec succès sur une carte à puce.

Trusted Logic X. Leroy propose lui aussi un BCV optimisé pouvant être embarqué sur une carte à puce [107]. Le procédé utilisé réalise une optimisation et une transformation du code à analyser. L'analyse de flot implémentée par ce BCV est alors suffisamment simple pour pouvoir être embarquée. X. Leroy a effectué une preuve préliminaire en Coq de la correction de la méthodologie utilisée.

E. Gimenez et al. présentent, dans [74], une formalisation importante et complète de la plate-forme JavaCard en Coq. Le module de liaison (linker), le BCV, une machine défensive, une machine offensive, deux machines construites comme des abstractions de la machine défensive : une machine vérifiant le typage et une machine vérifiant les propriétés assurées par le pare-feu. Les preuves de correction de chaque composant sont construites. De plus un BCV est implémenté, comme pour le projet Bali et CertiCartes, comme une variante de l'algorithme de Kildall. Ce travail représente un effort très important de formalisation. La plupart des aspects pertinents et complexes (qui sont, dans beaucoup d'autres formalisations, abstraits, ou laissés de coté) de la JCVM sont pris en compte par au moins une des variantes de la JCVM définies. Les auteurs indiquent un travail de 6 ans/homme et de 2 à 5 experts en continu sur la formalisation. Ils rappellent malgré tout que l'utilisation de systèmes comme Coq pour la vérification formelle d'applications industrielle est démontrée faisable, même si elle a encore un coût élevé. Leurs conclusions abonnent en faveur de la construction d'outils aidant (et systématisant) la construction de larges formalisations. Pour conclure ajoutons que malheureusement la formalisation n'est pas disponible, ce qui est un inconvénient majeur à la visibilité du projet.

Autres projets Si les travaux jusqu'ici présentés ont introduit des modèles et des vérifications de la plate-forme considérée comme un tout, une multitude de formalisations focalisées sur un point précis de la plate-forme ont été réalisées. Citons Bertot [28] qui a proposé une étude du problème de l'initialisation en Coq, d'après les travaux de Freund et Mitchell [66]. Citons aussi Jensen et Ségouat [140] qui ont formalisé en Coq la phase de conversion.

Z. Qian, A. Coglio et A. Goldberg proposent dans [48] une formalisation de la vérification de bytecode en SPECWARE. Une implémentation du BCV a été dérivée par raffinement. Cependant aucune preuve formelle n'a été développée. D. Syme [155] reprend les travaux de Drossopoulou [58] sur une sémantique opérationnelle et un système de types du langage source Java et prouve la sûreté du typage avec l'outil DECLARE [154]. Syme rapporte d'ailleurs que le simple fait de passer d'une formalisation papier à une formalisation machine a permis de corriger un certain nombre d'ambiguïtés et même d'erreurs. Toujours autour de la propriété de correction de typage, mais à propos de la plate-forme d'exécution CLR[52, 112], D. Syme et A. Gordon [156, 78] proposent une formalisation d'un sous ensemble conséquent du langage propre à cette plate-forme (IL, pour

Intermediate Language). A bien des égards, leur architecture pour la vérification formelle est proche de celle que nous présenterons dans les chapitres 4 et 5. D'une part, l'accent est mis sur l'exécutabilité des spécifications produites. D'autre part des techniques de preuve sont développées afin de permettre une meilleure automatisation de correction de typage. Il s'agit, dans le cadre de cette formalisation, de pouvoir réduire des énoncés complexes vers des énoncés démontrables par l'outil de preuve utilisé (le système de premier ordre SVC). Soulignons que ce projet ne met pas l'accent sur la dérivation du vérifieur de type à partir de la plate-forme défensive définie.

3.4.2 Vérification des applications Java/JavaCard

Il n'existe non pas une mais plusieurs approches à la vérification des applications Java et JavaCard. Chacune d'elles apporte ses méthodologies et ses outils pour atteindre des objectifs bien déterminés.

Par exemple :

- la vérification de programmes au niveau des sources permet d'aider les développeurs à assurer que leurs applets soient correctes. Le point fort d'une telle approche est que le développeur, ou un expert en méthodes formelles participant au développement, peut fournir une spécification formelle du programme et vérifier/prouver que l'applet est correcte par rapport à sa spécification. Il existe une grande diversité de formalismes permettant l'expression de telles spécifications. Les logiques des prédicats, temporelles ou dynamiques, ou encore les langages de spécification comme JML [93] sont des exemples de tels formalismes et les techniques de vérification associées incluent la vérification pendant la phase d'exécution, le test d'unité [44], les tests statiques [104], la vérification de modèles [51], les démonstrateurs de théorèmes interactifs [26] ;
- la vérification des programmes compilés, c'est-à-dire du bytecode, peut aider à vérifier s'il est sûr de charger et d'exécuter une applet dont l'origine est inconnue ou douteuse. Comme il n'est pas raisonnable de déléguer au consommateur la tâche de vérifier les applets chargées sur sa carte, il est important que ce type de vérification utilise des techniques automatiques comme les analyses statiques, la vérification de modèles [31, 129], ou s'appuie sur un schéma dans lequel la phase de vérification est déléguée au producteur de code [122].

Nous considérons dans cette section deux classes de travaux : ceux dont l'approche centrale est algorithmique et ceux dont l'approche est logique.

Approches Logiques

Loop Tool Le système Loop Tool [130] propose des outils pour raisonner sur des programmes écrits en langage source Java. Le Loop Tool présente une architecture construite autour des assistants de preuve PVS et Isabelle. Celui-ci a été utilisé avec succès pour la vérification de certaines APIs Java et plus récemment pour la vérification d'APIs JavaCard. L'approche Loop Tool s'articule autour des points forts suivant :

- une sémantique des programmes Java et du modèle de la mémoire, basée sur la théorie des types ;
- une logique axiomatique pour raisonner sur les programmes et la technologie Java/JavaCard. Les exceptions, terminaisons brutales, et l'héritage sont pris en compte par cette logique ;
- un outil de compilation de classes Java en théories PVS ou Isabelle. Ces théories forment le modèle sur lequel raisonner.

Une différence entre ces travaux et les nôtres est que leur sémantique traite des programmes en langage source et n'en permet pas l'exécution directe.

Krakatoa L'équipe du LRI propose un outil, Krakatoa [99], dérivant à partir d'un programme en langage source et d'annotations JML de celui-ci, un modèle et des obligations de preuve. Ces obligations de preuve sont les propriétés que le modèle extrait doit vérifier. PVS et Coq sont les assistants de démonstrations cible de la dérivation. L'outil en est encore à un stade préliminaire. Certaines annotations JML, les plus complexes, ne sont pas encore supportées, et des solutions à certaines difficultés liées aux traits impératifs sont étudiées.

Autres projets Des travaux semblables ont été développés par A. Poetzsch-Heffter, P. Müller et J. Meyer [113, 114, 128] qui proposent aussi une sémantique axiomatique d'une partie du langage Java et un support pour la vérification assistée (via PVS).

P. Brisset [36] propose une combinaison de techniques logiques et algorithmiques pour verifier la correction du gestionnaire de sécurité de Java. Citons finalement les projets suivants :

- le projet KEY [2], qui propose une intégration de modules de preuve dans un système d'aide à la conception. Ce module permet l'annotation des programmes par une logique dynamique (une extension de la logique de Hoare), génère et prouve (éventuellement interactivement) les obligations de preuve en résultant. À ce propos, une logique dynamique pour JavaCard est proposée [25] pour pouvoir supporter les annotations et les preuves sur les programmes JavaCard ;
- et finalement le projet CardKt [77] qui propose une vérification optimisée des programmes par une logique modale. La compacité du vérifieur permet d'envisager son embarquement sur une carte.

Approches Algorithmiques

Bandera L'équipe de J. Hatcliff et M. Dwyer a développé un ensemble d'outils regroupés sous le nom de Bandera [12, 87, 51]. Ceux-ci permettent de vérifier des programmes Java en langage source via vérification de modèles. Bandera a ainsi permis de vérifier des propriétés de programmes Java liées au multithreading. L'architecture de Bandera peut se résumer par :

- Un langage de spécification basé sur une logique temporelle pour l'expression de propriétés de programme ;
- un ensemble d'outils qui extraient du source Java des modèles à états finis compacts ; Ces outils implémentent plusieurs techniques d'analyses/trans-

formation de programmes, incluant l'évaluation partielle et le *program slicing* [163] ;
- un langage de spécification d'abstraction et un moteur d'abstraction permettant la dérivation de programmes abstraits. L'abstraction est ici utilisée afin de construire des modèles compacts essentiels à la vérification de modèles.

Autres projets D'autres équipes de recherche ont développé des travaux autour de cet axe algorithmique. Citons le laboratoire de recherche de GEMPLUS [31] qui a proposé une méthode basée sur l'abstraction et la vérification de modèles pour l'analyse de la transitivité du flot de données dans le cadre des cartes à puce.

Citons aussi les travaux de T. Jensen, D. Le Métayer et T. Thorn [92] qui ont proposé un cadre algorithmique à la vérification de propriétés de sécurité basée sur le flot de contrôle. Cette méthodologie a été reprise et mise à profit pour la vérification compositionnelle de propriétés liés au flux de données de programmes JavaCard [23].

Pour finir, mentionnons K. Havelund [88] qui, dans le cadre du projet "Java Path Finder" propose une vérification de modèles de programmes Java concurrents. Puis K. R. M. Leino, J.B. Saxe et R. Stata [105] qui proposent une vérification statique interfacée avec un démonstrateur de théorème (SIMPLIFY).

3.5 Conclusion

L'utilisation des méthodes formelles a évolué très rapidement ces dernières années. Si l'utilisation des outils de preuve n'était pas viable il n'y a pas si longtemps, d'énormes efforts de conception ont permis d'introduire les systèmes de spécifications, puis les démonstrateurs de théorèmes et finalement les assistants de preuve dans le processus de formalisation et de validation.

Nos conclusions à cet égard rejoignent celles de Gimenez [74] et de Nipkow et von Oheimb qui, dans [125], affirment :

> *machine-checking the design of a non-trivial programming language has become a reality*

L'utilisation actuelle des méthodes formelles permet enfin d'envisager les niveaux EAL 5, 6 et 7. Pour ce dernier niveau rappelons que la portée et le grain de la spécification sont importantes (se reporter à la section 3.2). En ce sens, CertiCartes est une formalisation complète de la plate-forme JavaCard qui constitue une étude de faisabilité approfondie d'évaluation EAL 7 [101].

Cependant, les efforts de formalisation de Trusted Logic[74] et ceux de J. Strother Moore [118] les ont tout de même amené à formuler quelques réticences sur la vérification assistée par ordinateur de larges spécifications :

Le problème d'échelle et de complexité. La spécification complète de la plate-forme JavaCard implique un niveau de détail (plus le grain est fin plus les détails sont importants) qui nuit, de fait, à la clarté de la spécification mais qui est inévitable sans outils appropriés et/ou méthodologies robustes. La finesse de la spécification est malheureusement aussi un facteur important dans la difficulté de démonstration. La démonstration n'en est que plus complexe et doit prendre en compte beaucoup de détails non pertinents ;

Le problème d'expertise et de lourdeur. La remarquable formalisation réalisée par Trusted Logic représente un effort très important. Pour prendre en compte les problèmes de foisonnement de détails, plusieurs abstractions ont été implémentées. Il s'agit, malgré tout, d'un lourd travail d'expert : six ans/homme ont été nécessaires, deux à cinq *experts* ont travaillé en parallèle sur la formalisation.

Trusted Logic préconise la constructions de nouveaux outils permettant de prendre en compte les lourdeurs propres à la taille de la tâche (de spécification et de preuve).

JaKarTa est notre contribution à la résolution de ce problème :

– JaKarTa apporte des outils permettant d'assister le concepteur face aux problèmes de granularité. Dériver une spécification possédant une granularité différente est l'une des fonctionnalités supportées ;
– JaKarTa apporte des outils de preuve aidant la démonstration de propriétés de spécification JaKarTa.

Soulignons la pertinence de ces deux points. JaKarTa se propose de fournir des outils permettant de gérer de façon *systématique* les problèmes de profusion de détails et de garantir, tout aussi systématiquement, la correction des transformations réalisées.

Chapitre 4

CertiCartes : une sémantique formelle exécutable JavaCard

Nous abordons dans ce chapitre la construction de la spécification formelle exécutable d'un composant essentiel de la plate-forme JavaCard : la machine virtuelle JavaCard (JCVM). Ainsi, nous définissons une sémantique à petit pas de plusieurs machines virtuelles :

- la première, appelée défensive (notée dJCVM), réalisant des vérifications de typage pendant l'exécution, comme implicitement suggéré par la spécification SUN [152, 151] ;
- la deuxième, appelée offensive (notée oJCVM), n'exécute pas de telles vérifications et repose sur la vérification de bytecode pour le bon typage ;
- finalement, nous construisons une machine virtuelle abstraite (aJCVM) qui opère sur les types plutôt que sur les valeurs. Une telle machine est la base de la construction du vérificateur de bytecode (BCV) analysant le bon typage d'un programme (voir [17]).

Ensuite, nous fournissons les preuves formelles de correction :

1. pour les machines virtuelles offensive et défensive, nous montrons que les deux machines coïncident sur les programmes dont l'exécution sur la dJCVM ne lance pas d'erreur de type ;
2. nous montrons aussi que tous les programmes dont l'exécution sur la dJCVM lance une erreur de type lancent aussi une erreur de type lorsqu'ils sont exécutés par la aJCVM.

Ces démonstrations constituent la validation croisée des différentes machines virtuelles définies et forment une composante essentielle à la preuve de correction du BCV dérivé de la machine abstraite.

De plus, notre sémantique formelle est exécutable. Ainsi, n'importe quel programme JavaCard peut être exécuté et le comportement de la machine virtuelle peut être comparé à celui des implémentations de référence, comme la machine virtuelle de SUN. L'exécutabilité constitue un argument de poids à la fiabilité d'une formalisation et, de notre point de vue, toute sémantique opérationelle formelle d'un langage (réaliste) doit être exécutable. Nous reviendrons sur ce point section 4.5.

Organisation du chapitre Dans la section 4.2, nous décrirons la formalisation des programmes JavaCard après liaison, et dans la section 4.3 nous décrirons le JCVM Tools qui transforme les fichiers au format CAP en leur représentation Coq. Dans la section 4.4 nous décrirons une sémantique à petit pas de la machine virtuelle défensive JavaCard, où chaque instruction est modélisée comme un modificateur d'état. Dans les sections 4.7 et 4.8 nous aborderons la dérivation des machines offensive et abstraite à partir de la machine défensive et nous en montrerons la correction : aussi bien la construction que la preuve s'expriment dans le cadre de l'interprétation abstraite. Celle-ci fournit une théorie sémantique bien établie pour ce propos. Nous aborderons ce point à la section 4.6. Nous conclurons section 4.10 sur une analyse et les possibles extensions de CertiCartes.

4.1 Considérations préliminaires

La formalisation présentée ici est une formalisation à deux niveaux : (1) la spécification de la sémantique opérationnelle des machines virtuelles ; (2) l'ensemble des preuves de la correspondance entre les différentes machines virtuelles.

La sémantique des différentes machines virtuelles est exprimée en terme de transformateurs d'état. Leur formalisation utilise un style mathématique neutre basé sur les types de données, enregistrements, fonctions récursives ; en particulier notre formalisation n'utilise ni les types dépendants ni les fonctions d'ordre supérieur. Ce choix de conception est motivé par la volonté d'obtenir une spécification portable et exécutable. De fait, la spécification présentée ici a été traduite (consulter à ce propos le chapitre 5) vers d'autres plates-formes.

Nous avons aussi tenté de rendre notre formalisation aussi lisible que possible. Nous avons ainsi factorisé autant que possible la spécification au niveau de la sémantique opérationnelle des bytecodes : plusieurs bytecodes ne diffèrent seulement que par les types de données sur lesquels ils agissent et sur le type d'opérations qu'ils effectuent sur celles-ci. Une illustration de ce dernier point sont les instructions arithmétiques qui procèdent de façon identique pour obtenir les entrées pour l'opération considérée puis pour délivrer la sortie calculée (voir 4.4.4).

Ces considérations nous ont donc ainsi amené à être aussi modulaire que possible, en particulier à spécifier plusieurs bytecodes par une seule fonction Coq.

Selon les mêmes principes, bien que tous les indices utilisés dans la spécification soient tous des nombres naturels et aient besoin d'être traités comme tels, nous définissons des synonymes de `nat` pour les différents types d'index présents dans la formalisation. Par exemple, les index de classes dans les fichiers CAP sont définis en Coq par :

```
Definition cap_class_idx := nat.
```

Nous verrons au chapitre 6, dans le cadre de l'abstraction de certains de ces types, comment ce choix s'est révélé judicieux.

Le style fonctionnel choisi n'est pas une option commune dans le domaine de la vérification assistée de la sémantique des langages de programmation. Il est même frappant de voir que beaucoup de travaux semblables choisissent d'adopter le style relationnel dans lequel les spécifications sont écrites “à la

Prolog" en utilisant les relations inductives. Cependant nous avons préféré le style fonctionnel pour les raisons suivantes :

- Premièrement, ce style permet une plus grande réutilisation de la spécification. De fait, nous avons réussi à porter CertiCartes vers d'autre plateformes que Coq, comme PVS [142], OCaml [108], Elan [32], JaKarTa [20] et Spike [34] ;
- Deuxièmement, les spécifications écrites dans un style fonctionnel sont exécutables. Comme nous l'avons signalé dans l'introduction, cette propriété nous permet (1) de vérifier et de comparer notre spécification avec les implémentations de référence. Cette comparaison est cruciale pour s'assurer de la correction de la spécification ; (2) de produire les traces d'exécution qui peuvent être utilisées pour pouvoir raisonner sur le programme exécuté. Toujours à ce propos, les spécifications exécutables permettent une automatisation des preuves car elles offrent la possibilité d'exploiter les tactiques de réécriture, comme l'illustre l'article [16].

La machine défensive La première machine que nous allons définir est une spécification fidèle aux documents de références de JavaCard [151, 152] en ce sens qu'elle implémente toutes les pré-conditions (les clauses *must* et *must not* de la spécification officielle de SUN) à l'application des instructions.

4.2 Représentation des programmes JavaCard

Dans cette section nous présentons la formalisation Coq des programmes JavaCard. Comme nous l'avons déjà indiqué, notre formalisation traite des programmes manipulant du bytecode. Ces programmes correspondent au code compilé qui est exécuté par la machine virtuelle JavaCard. De façon à proposer une description complète de la sémantique opérationnelle de JavaCard (section 4.4), nous avons besoin premièrement de définir les structures de données manipulées par les programmes et alors de donner une représentation des programmes avec leurs classes, méthodes et interfaces.

4.2.1 Structures de données et valeurs

La machine virtuelle JavaCard distingue les types primitifs des types références (comme par exemple les références aux instances de tableaux, de classes ou encore d'interfaces). Les types primitifs sont soit des types arithmétiques (`Byte`, `Short`, `Int`) soit le type des booléens (`Boolean`), soit le type des adresses de retour (`ReturnAddress`). Pour être plus précis, `ReturnAddress` est le type des valeurs qui représentent une référence à un point du programme vers lequel il faudra se rendre, et ceci dans le cadre de l'exécution d'une sous-routine. De telles valeurs sont manipulées par les bytecodes `jsr` et `ret` qui, respectivement, indiquent qu'il faut exécuter (sauter vers) une sous-routine et revenir après le point d'appel de la sous-routine. Notons aussi l'existence du type spécial non dynamique `Void` qui signale le type sans valeur.

Formellement les types primitifs JavaCard sont introduits en Coq par la définition suivante :

```
Inductive type_prim : Set :=
  Byte          : type_prim |
  Short         : type_prim |
  Int           : type_prim |
  Int2          : type_prim |
  Boolean       : type_prim |
  Void          : type_prim |
  ReturnAddress : type_prim.
```

Notons que `Int` a une variante `Int2`. Ceci est dû au fait que le type JavaCard `Int` utilise deux cellules mémoire. La première cellule mémoire signifiant une valeur entière est donc marquée par le tag `Int` alors que la deuxième cellule l'est avec `Int2`. Ce détail technique nous permet de prendre en compte l'extraction complète d'un entier (c'est-à-dire de la valeur entière contenue dans deux cellules mémoire) ainsi que le bon comportement de bytecodes comme `dup2` qui duplique une valeur qui occupe deux cellules mémoire.

Les références sont de quatre types : les références nulles, les références de tableau, les références d'instance et les références d'interface. Les tableaux se définissent par rapport au type de leurs éléments. Les références d'instance ou d'interface sont représentées par des entiers naturels. Ces entiers sont en fait la position dans la liste des classes ou dans la liste des interfaces où se trouvent les classes ou interfaces considérées (Section 4.2.3). Ainsi, les types JavaCard sont définis de façon mutuellement inductive par :

```
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  : cap_class_idx → type_ref   |
  Ref_interface : cap_interf_idx → type_ref.
```

Ici nous avons introduit le type référence `type_ref` qui, par le biais du constructeur `Ref_array` se réfère à `type`. C'est ce fait qui justifie la définition mutuellement inductive. Ce constructeur représente le type des tableaux d'éléments du type de son argument (qui peut être une référence ou un type primitif). À ce niveau, les tableaux de tableaux sont formulables, ce qui n'est pas permis dans JavaCard. C'est la sémantique opérationnelle des bytecodes permettant la création de tableaux qui interdira la création de tels tableaux.

4.2.2 Les valeurs dynamiques

Étant données la représentation des types JavaCard et la nature défensive de la machine virtuelle que nous désirons construire, les valeurs des programmes sont typées, c'est-à-dire que leur représentation comprend, outre la valeur numérique, l'information du type dynamique (du type durant l'exécution). Celles-ci peuvent être des valeurs arithmétiques ou non (comme par exemple les adresses mémoire). Nous avons choisi de représenter chacune d'entre elles comme un entier. Dans le cas des valeurs non-arithmétiques nous établissons une coercion implicite vers les entiers. Ainsi, comme dans [131], nous marquons les valeurs par leur type. Remarquons que la représentation des types dynamiques est une simple adaptation des types précédemment introduits. Formellement nous avons : [1]

```
Inductive valu_prim : Set :=
  vReturnAddress : bytecode_idx → valu_prim |
  vBoolean       : Z → valu_prim |
  vByte          : Z → valu_prim |
  vShort         : Z → valu_prim |
  vInt           : Z → valu_prim |
  vInt2          : Z → valu_prim.

Inductive valu_ref : Set :=
  vRef_null      : valu_ref |
  vRef_array     : type → heap_idx → valu_ref |
  vRef_instance  : cap_class_idx → heap_idx → valu_ref |
  vRef_interface : cap_interf_idx → heap_idx → valu_ref.

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

Le choix d'un type inductif pour représenter les valeurs typées plutôt qu'un type produit $type * Z$ est dû au souci d'automatisation du processus de dérivation de machine virtuelles (voir chapitre 5). Les premières versions de CertiCartes considéraient d'ailleurs la définition suivante de `valu` : **`Definition`** `valu := type*Z.`

Le type `Z` est le type des représentations binaires d'entiers tel qu'il est défini dans les bibliothèques Coq. Dans le cas des valeurs de type `Int`, qui sont contenues dans deux cellules mémoire contiguës de type respectifs `Int` et `Int2`, nous choisissons d'associer à chaque cellule la même valeur numérique. Ceci évite d'avoir à considérer, pour la valeur en question et pour sa représentation en mémoire, la séparation des bits de poids fort des bits de faible poids. Cet état de fait ne change en rien le comportement calculatoire de ces valeurs, et rend même le raisonnement plus aisé (chaque entier a une représentation unique dans `Z`).

Notons cependant que nous fournissons une représentation alternative des valeurs en mémoire qui est plus proche de ce que l'on trouve dans les implémentations de machines virtuelles. Cette représentation déclare les entiers comme des listes de bits `Z_bits`. Ce type nous permet de définir toutes les opérations arithmétique comme des fonctions sur `Z_bits` où on peut prendre en compte les questions de débordement de capacité telles qu'elles sont définies dans la spécification SUN. Finalement, pour palier les problèmes dus à la dualité de la représentation des entiers, nous avons défini la coercion bidirectionnelle entre `Z` et `Z_bits` permettant de passer d'une représentation à l'autre.

[1] L'identificateur Coq `value` étant déjà utilisé, nous utilisons plutôt l'identificateur `valu`.

4.2.3 Représentation des programmes

Un programme JavaCard met en jeu des classes, des méthodes et des interfaces. Les limitations de la mémoire et de la capacité de calcul d'une carte à puce obligent les programmes JavaCard à ne pas prendre en compte toutes les possibilités offertes par Java (voir section 2.4). Malgré cela, la modélisation des classes reste identique. L'appel de méthodes, la définition et l'implémentation des interfaces sont les mêmes que celles d'un programme Java.

Un ensemble particulier de ces éléments sont les classes, les méthodes, les interfaces et les instances du package `java.lang`. Parce que nous avons choisi de manipuler des programmes clos, ceux-ci font partie intégrante de tout programme dans notre formalisation. Ceci inclut en particulier la classe `Object`, l'interface `Shareable`, et les classes des exceptions.

La définition suivante formalise la représentation des programmes :

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

Les types `Interface`, `Class` et `Method` sont eux-mêmes définis comme des enregistrements.

Nous pouvons inférer à partir de l'analyse du programme, la liste des types des champs statiques déclarés. Cette analyse est faite automatiquement par le JCVM tools (section 4.3) et le résultat est placé dans le champ `static_heap_type`.

Les Classes

Dans le langage Java, une classe représente une certaine catégorie d'objet. Une classe définit des variables (attributs), des méthodes aussi bien que les interfaces qu'elle implémente et sa filiation (si celle-ci n'est pas précisée, la classe `Object` sera implicitement la super-classe).

Dans notre représentation, une classe est décrite par la donnée :
- de ses super-classes (le cas échéant),
- de ses méthodes (constructeurs compris),
- des interfaces qu'elle implémente agrémentée des adresses des méthodes qui en représentent l'implémentation,
- du type des variables de classe,
- du contexte propriétaire,
- finalement de l'identificateur de la classe elle-même. C'est l'index de la classe dans la liste des classes qui est pris comme identificateur.

Notre description d'une classe `c` se réfère à toutes les classes desquelles `c` hérite, c'est-à-dire à toutes les classes en relation avec `c` par la clôture transitive de la relation d'héritage direct. Pour des questions d'efficacité – en partie pour éviter la définition de fonctions bien fondées par la relation de sous-classe, toutes ces classes sont collectées dans une liste et par convention le premier élément de cette liste est la classe mère. Ce qui donne en Coq :

```
Record Class : Set := {
                    (* Liste de toute les super classes de la classe *)
  super           : (list cap_class_idx);
                    (* Liste des methodes publiques ou protegees *)
  public_methods  : (list Method);
                    (* Liste des methodes privees ou package *)
  package_methods : (list Method);
                    (* Liste des interfaces implementees avec leur methodes *)
  int_methods     : (list cap_interf_idx*(list class_pbmethod_idx)));
                    (* Liste des types des variables de classe *)
  class_var       : (list type);
                    (* Identification du package contexte proprietaire *)
  package         : Package;
                    (* Indice de la classe *)
  class_id        : cap_class_idx
}.
```

où `cap_class_idx` et `cap_interf_idx` correspondent aux types des indices dans le fichier CAP des classes et des interfaces. Le type `class_pbmethod_idx` correspond au type des indices des éléments du champs `public_methods`. Tous ces indices sont des entiers naturels, c'est-à-dire que tous les types d'indices sont des synonymes du type Coq `nat` .

Le langage JavaCard distingue plusieurs types de méthodes. Ainsi les méthodes publiques et protégées sont séparées des méthodes package et privées. Cette distinction existe déjà dans les fichiers CAP. Il est important d'ajouter que toute méthode d'interface indicée dans le champ `int_methods` est membre soit de `public_methods` soit de `package_methods`.

Remarquons que notre représentation ne prend pas en compte la profondeur maximale de la pile d'opérande pendant l'exécution. L'ajout d'une telle information ne soulève cependant pas de difficulté.

Penchons-nous maintenant sur le type `Package`. À l'image de la plate-forme Java, un package est un groupe de classes et d'interfaces qui constituent un programme.

Les packages sont utiles au contrôle d'accès aux classes, aux champs statiques et aux méthodes statiques. Dès lors, pour des raisons de sécurité toute classe appartient à un seul package. Ce point est particulièrement important lorsque l'on est dans un cadre ou plusieurs applications coexistent.

Le regroupement de classes simplifie la communication entre applets appartenant au même package et permet un contrôle des communications dans le cas contraire. Les packages sont identifiés par ce qu'on appelle un IDentificateur d'Application (AID) qui est la chaîne numérique permettant de les identifier. Dans notre formalisation nous considérons les AIDs comme des listes d'entiers naturels. Ainsi, le type `Package` est défini par :

```
Definition Package := (list nat).
```

Les valeurs de ce type joueront le rôle de clés de sécurité dans l'architecture sécuritaire de JavaCard (voir section 4.4.4)

Interfaces

Les interfaces définissent une collection de signatures de méthode et de constantes. Celles-ci doivent être définies dans toute classe implémentant ces interfaces. Ce concept d'interface est à rapprocher de la notion de classes abstraites et correspond, dans l'approche Java(Card), à une alternative à l'héritage multiple.

Concrètement, la plupart de l'information contenue dans une interface n'est pas pertinente dans le cadre de notre formalisation. Une grande partie de l'information n'est pertinente que pendant la phase de liaison (voir section 4.3). En effet, c'est le lieur qui vérifie qu'une classe implémentant une interface le fait de façon correcte. Cette phase de vérification est donc réalisée à un niveau qui ne concerne pas notre formalisation. Nous ne considérons que les programmes ayant passé l'étape de liaison. Dès lors, nous n'avons besoin que de garder l'information des super-interfaces d'une interface, pour nous permettre par exemple de vérifier la compatibilité de deux vecteurs d'interfaces. Ainsi nous avons :

```
Record Interface : Set := {
             (* Liste de toutes les super interfaces *)
  super_int : (list cap_interf_idx);
             (* Indice de l'interface *)
  int_id     : cap_interf_idx
  }.
```

Il existe une interface définie dans les APIs de JavaCard qui revêt une importance particulière dans l'architecture sécuritaire de la plate-forme. Cette interface, appelée `Shareable`, permet le conditionnement des communications entre les applets. Concrètement, une interface i est dite *partageable* si et seulement si c'est l'interface `Shareable` ou si celle-ci est une super-interface de i (i.e. membre de `super_int i`)

Méthodes

Les méthodes sont des fonctions définies dans une classe. Une méthode est caractérisée par :

- son statut (statique ou non),
- sa signature (qui peut servir, par exemple, à la vérification du typage des paramètres qui lui sont passés),
- le nombre de variables locales qu'elle possède (utile à l'initialisation de son contexte d'exécution),
- ses récupérateurs d'exception (ceux-ci comprennent le code nécessaire à la gestion des exceptions),
- le code même définissant la méthode (présenté sous la forme d'une liste d'instructions)
- et finalement les indices relatifs à la méthode elle-même (indice de la méthode dans la liste des méthodes du fichier CAP et à la classe à laquelle elle appartient).

Nous avons donc :

```
Record Method : Set := {
```

```
                      (* signature de la methode *)
signature           : (signature_type*type);
                      (* nombre variables locales *)
local               : nat;
                      (* la liste des instructions *)
bytecode            : (list Instruction);
                      (* statut de la methode *)
statusstatic        : bool;
                      (* listes des recuperateurs d'exception *)
handler_list        : (list handler_type);
                      (* indice de la methode *)
method_id           : cap_method_idx;
                      (* indice de la classe proprietaire *)
owner               : cap_class_idx
}.
```

où le type `handler_type` regroupe l'information nécessaire au calcul du meilleur candidat à la récupération d'une exception donnée à un point donné du programme. Nous utilisons la structure suivante pour représenter ce type :

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

Les deux premiers éléments définissent la portion du code dans lequel le récupérateur en question peut agir, le troisième élément définit la classe de l'exception dont s'occupe le récupérateur considéré, et le dernier indique où commence le code à exécuter si ce récupérateur est sélectionné.

Le meilleur récupérateur est calculé à partir de la liste des candidats regroupés dans `handler_list` et du type de l'exception levée. La sélection a lieu selon un procédé que nous établirons dans la section 4.4.3.

Pour ce qui est de la signature, le type `signature_type` est simplement défini comme une liste de type :

```
Definition signature_type := (list type).
```

Comme considération finale à la représentation des programmes, remarquons que nous avons introduit une certaine redondance dans cette représentation. En effet, pour des raisons de contrôle et d'exécution, nous avons choisi de placer la même information à plusieurs endroits. Par exemple, une méthode se trouve aussi bien dans la description de la classe à laquelle elle appartient que dans la structure générale qui représente le programme, alors que nous aurions pu considérer uniquement garder l'indice de la méthode dans la structure représentant la classe (c'est-à-dire remplacer `Method` par `cap_method_idx` dans les champs appropriés du type `Class`). L'option choisie nous permet de manipuler directement la méthode au lieu d'avoir systématiquement à la chercher puis à la charger. La vérification de la correction d'une duplication est assurée par l'outil appelé *JCVM Tools*.

4.3 Le JCVM Tools

Comme nous l'avons déjà souligné, l'exécutabilité est, à notre avis, une caractéristique importante pour les spécifications formelles. Notre sémantique formelle est exécutable en ceci que chaque programmes JavaCard est représenté par une expression Coq de type `jcprogram` qui peut être évaluée. Mais il est évident que cet aspect prend toute son ampleur lorsqu'il est possible de traduire et exécuter n'importe quel programme JavaCard dans le système Coq. Ainsi, nous devons être capable de transformer un ensemble de fichiers CAP en une expression Coq (de type `jcprogram`) représentant l'intégralité d'un programme contenu dans ces fichiers. Cette traduction est assurée par un programme écrit en Java appelé *JCVM Tools*. Cet outil exécute cette conversion de même qu'il impose certains invariants sur tous les éléments de type `jcprogram` générés.

Les transformations effectuées par le JCVM Tools ainsi que les invariants imposés comprennent :

– la résolution complète, l'élimination de la table des constantes (*constant pool* dans la terminologie Java(Card)) et la liaison. Malgré le fait que notre formalisation ait été construite dans le but de pouvoir exécuter n'importe quel programme JavaCard, nous ne travaillons pas directement avec la représentation en flux de bits du format CAP, mais avec une représentation structurée de celui-ci. Celle-ci est obtenue par la reconstruction de la structure du programme (semblable à celle construite par un compilateur) avant la sérialisation à partir de laquelle les fichiers CAP sont créés.
  L'intérêt de cette phase est de deux ordres :

  1. Elle nous permet une représentation concise et claire du format CAP ainsi qu'une spécification de la sémantique opérationnelle du bytecode plus simple à mettre en œuvre ;

  2. L'intégration d'une formalisation d'un compilateur à notre formalisation en est facilitée dès lors que la correspondance entre la sortie d'un tel compilateur et l'entrée de notre formalisation est presque directe : notre format des programmes est très structuré ;

– la transformation des adresses relatives en adresses absolues, par exemple dans les instructions de branchements ;
– la synthèse de certains paramètres qui sont utilisés dans notre sémantique mais qui ne sont pas présents dans la spécification SUN. Par exemple le bytecode `invokevirtual` prend normalement en paramètre un indice vers la table des constantes qui se résout en une référence à une classe et en une méthode virtuelle. Dès lors que ce composant n'est pas présent dans notre formalisation, cet indice est remplacé par deux opérandes : l'indice de la classe et l'indice de la méthode.

De plus le JCVM Tools assure un nombre d'invariants qui sont requis par la spécification SUN, par exemple :

– l'invocation de méthode n'a recours qu'à des indices de méthodes qui sont plus petits que le nombre de méthodes existantes dans le programme ;
– l'invocation de méthode respecte les contraintes de visibilité des méthodes invoquées. Ainsi l'outil vérifie, entre autres, que les méthodes privées ne sont pas invoquées hors de leur package ;
– de même, l'invocation de méthodes statiques ne s'applique qu'à des méthodes statiques ;
– les adresses cible des instructions de branchements pointent bien vers des

instructions contenues dans la méthode appelée ;
- l'ordre de placement des récupérateurs d'exception inspectant la même portion de code est respecté par la transformation. En d'autres mots cela signifie que dans la liste de récupérateurs d'exception résultant de la transformation du code suivant,

```
try {...}
catch ( C1_ex x1 ) {...}
catch ( C2_ex x2 ) {...}
```

le récupérateur représentant `C1_ex` sera placé avant celui de `C2_ex`. Ce point subtil a son importance, considérons par exemple que `C2_ex` soit une sous-classe de `C1_ex` dès lors le récupérateur associé à `C2_ex` ne sera jamais sélectionné.

Puisque ces invariants sont imposés par le JCVM Tools, notre sémantique les considère comme acquis et ne les vérifie donc pas. Notons cependant que leur vérification ne pose pas de problème particulier.

4.3.1 Correction de la représentation

Notre représentation des programmes JavaCard introduit quelques redondances afin d'éviter d'avoir recours à de trop nombreuses fonctions définies par récursion bien fondée qui affectent souvent aussi bien l'efficacité de l'exécution que la complexité de raisonnement. Il faut cependant s'assurer de la cohérence de la duplication. Comme nous avons choisi de ne pas utiliser les types dépendants, nous devons nous reposer sur les prédicats de correction qui testent si un habitant du type `jcprogram` représente ou non un programme JavaCard. Peu de tests de correction sont nécessaires à l'établissement de la correspondance entre machines virtuelles, mais cette vérification de la correction est essentielle dans la construction d'un vérificateur de bytecode correct (consulter la référence [17]).

4.4 La Machine Virtuelle Défensive

La machine virtuelle est décrite par une sémantique à petit pas ; plus précisément, chaque instruction est formalisée comme un transformateur d'état. Un tel transformateur d'état est une fonction prenant en entrée un état de la machine virtuelle — c'est-à-dire une configuration dans laquelle la machine virtuelle se trouve avant l'exécution d'une instruction, et retourne un nouvel état résultant du calcul produit par l'exécution de l'instruction.

Dans notre formalisation, cet état dépend donc de la configuration de la mémoire qui doit contenir tous les éléments manipulés par un programme JavaCard. Ces éléments sont présentés dans cette section.

Considérations préliminaires La formalisation de la machine virtuelle présentée dans cette section suit la spécification officielle définie par SUN [152]. C'est, au sens définie dans la section 4.6, une machine défensive. Pour permettre un traitement uniforme des définitions propre à chaque machine (voir sections 4.7 et 4.8) nous préfixerons les définitions propres à la machine défensive par la lettre *d*.

4.4.1 Le modèle mémoire

Les états de la machine virtuelle sont formalisés comme des enregistrements à trois champs, constitués du tas (contenant les objets créés pendant l'exécution), du tas statique (qui contient les valeurs des champs statiques des classes) et de la pile d'environnements d'exécution de méthodes. Formellement nous avons :

```
Record dstate := {
  hp : heap;
  sh : static_heap;
  sk : stack
}.
```

Afin de décrire les différents éléments constituant la mémoire, commençons par la notion d'objet stocké dans le tas.

Les Objets

Un objet, dans la technologie JavaCard, est soit une instance soit un tableau. Dès lors le type des objets est naturellement défini comme un type inductif :

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

où `type_instance` et`type_array` sont tous deux des types enregistrement contenant toute l'information nécessaire à la description des instances et des tableaux.

Par exemple, une instance de classe est décrite par l'indice de la classe en question, par ses variables d'instance (formalisées comme une liste de `valu`), par la référence au package propriétaire et par son statut (un booléen indiquant si l'instance est ou non une instance point d'entrée). Les points d'entrée sont impliqués, dans le modèle de sécurité JavaCard, dans le processus de contrôle d'accès. Ce statut permet aux méthodes d'une instance "point d'entrée" d'outrepasser certaines contraintes d'accès et d'être invoquées sans restriction. Il y a deux types d'instances "point d'entrée" : les points d'entrée temporaires sont des instances dont les références ne peuvent être réutilisées (ce contrôle incombe au pare-feu), les points d'entrée permanents sont des instances dont la référence ne souffre pas de cette restriction et peu donc être stockée dans des variables de classe, d'instance ou même dans des tableaux. Le JCRE (JavaCard Runtime Environment) implémente et se charge ainsi de garantir le modèle de sécurité de JavaCard, et donc en particulier vérifie la bonne utilisation des instances. Nous avons :

```
Record type_instance : Set := {
  reference                : cap_class_idx;
  contents_i               : (list valu);
  owner_i                  : Package;
  is_entry_point           : bool;
```

```
  is_permanent_entry_point : bool;
}.
```

Les tableaux sont définis de façon tout à fait semblable. Dès lors que tous les éléments d'un tableau ont le même type, ceux-ci n'ont pas besoin d'être du type `valu`, mais plutôt du type `Z`, évitant ainsi une redondance d'information. Ainsi, le type des éléments n'est stocké qu'une seule fois au sein de la représentation des tableaux. Ce qui donne :

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

Les deux derniers champs représentent respectivement le package propriétaire et le statut du tableau. En effet un tableau peut être global ou pas. Cette particularité est proche de celle des points d'entrée en ceci qu'elle permet d'attribuer quelques privilèges d'accès. Ainsi un tableau global est accessible de n'importe quel contexte.

Le Tas

Pendant l'exécution d'un programme, le tas contient les instances des classes et les tableaux. Le tas statique ne contient que les champs statiques. Le tas et le tas statique sont donc respectivement définis comme des listes d'objets et de valeurs :

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

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

La pile

La pile regroupe les environnements d'exécution (aussi appelés contextes d'exécution, ou frame dans leur terminologie anglo-saxonne) en cours des différentes méthodes appelées. Un environnement d'exécution est donc créé lors d'un appel de méthode et est détruit à la fin de l'exécution de la méthode appelée.

Formellement nous avons :

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

Le type `frame` déclare les éléments nécessaires à la formation des environnements d'exécution. Il regroupe ainsi la liste des variables locales, la pile d'opérande où sont stockés les paramètres et résultats, le compteur de programme indiquant à quel point du programme de la méthode l'exécution doit continuer (i.e. la prochaine instruction à exécuter), la référence à la méthode associée à l'environnement et finalement la référence au contexte propriétaire de la méthode. Cette dernière information joue un rôle important dans le mécanisme du pare-feu. Ainsi :

```
Record frame : Set := {
  locvars        : (list valu);       (* Variables Locales *)
  opstack        : (list valu);       (* Pile d'operande *)
  p_count        : bytecode_idx;      (* Compteur de Programme *)
  method_loc     : cap_method_idx;        (* Indice de la  methode *)
  context_ref    : Package;           (* Contexte proprietaire *)
}.
```

4.4.2 Les Instructions

Notre formalisation Coq des instructions traite celles-ci d'une façon assez uniforme. En effet, quelque soit l'instruction, l'exécution de celle-ci met à jour la mémoire de la machine virtuelle en accord avec les opérandes générées pendant la compilation et avec la représentation des programmes. Cependant, si nous nous penchons sur le type d'actualisation réalisée par chaque instruction nous pouvons quand même dégager les huit catégories d'instructions suivante :

- opérations arithmétiques (`sadd`, `idiv`, `sshr`...) ;
- tests sur les objets (`instanceof`...) ;
- branchements, éventuellement conditionnels (`ifcmp`, `goto`...) ;
- appels de méthode (`invokestatic`, `invokevirtual`...) ;
- instructions opérant sur les variables locales (`iload`, `sstore`...) ;
- instructions opérant sur les objets, ou plus précisément sur les tableaux ou les instances (`getfield`, `arraylength`...) ;
- instructions opérant sur la pile d'opérande (`ipush`, `pop`...) ;
- instructions de retour (`sreturn`, `throw`...).

Nous avons formalisé l'ensemble des instructions par le type inductif suivant : `Instruction` :

```
Inductive Instruction : Set :=
   nop    : Instruction |
   push   : type_prim → Z → Instruction |
   load   : type → locvars_idx → Instruction |
   aload  : type → Instruction |
   store  : type → locvars_idx → Instruction |
   astore : type → Instruction |
   ...
```

où chaque couple (constructeur, signature) représente la syntaxe des bytecodes considérés.

En reprenant la vision unificatrice du traitement des instructions, la sémantique de chaque instruction est formalisée par l'utilisation d'une fonction de type :

```
dstate * operands → rdstate
```

le type des opérandes étant, lui, spécifique à chaque instruction.

Une exécution peut se conclure de façon abrupte lorsque une exception lancée n'a pu être rattrapée. Ainsi, de façon à pouvoir considérer ces exécutions, l'image des fonctions codant la sémantique des instructions est le type inductif suivant :

```
Inductive rdstate: Set :=
```

```
Normal   :        dstate → rdstate |
Abnormal : xcpt → dstate → rdstate.
```

Dans le cas d'une exécution sans problème, l'état de retour est l'état résultant de l'exécution normale de l'instruction considérée, coiffé du constructeur `Normal`. Dans le cas contraire, l'état produisant l'erreur ainsi que le descriptif de l'exception non rattrapée sont retournés en utilisant le constructeur `Abnormal`.

Passée la phase de définition sémantique de chaque instruction, nous sommes en mesure de formuler ce qu'on entend par exécution de programme. La fonction `exec_instr` calculant un pas d'exécution a ainsi le type suivant :

```
instruction*dstate*program → rdstate
```

Elle prend en entrée une instruction `i`, un état `s` et un programme `p`. Elle retourne `sem_i s o` où `sem_i` est la fonction formalisant la sémantique de `i` (de type `dstate*operands → rdstate`) et `o` la liste des opérandes nécessaire à l'exécution de l'instruction, celles-ci étant extraites de `dstate` et de `program`.

Le choix de la formalisation de l'exécution par une fonction d'exécution pas à pas repose sur le fait que nous nous plaçons dans un cadre ou seules sont formulables les fonctions qui terminent. C'est effectivement le cas des fonctions définissables en Coq. Il n'est donc pas possible, en Coq, de définir la fonction générale d'exécution (la fonction qui exécute jusqu'à ce que le programme termine). Notons cependant que ce n'est pas une limitation contraignante. Nous avons dérivé en utilisant le mécanisme d'extraction de Coq la fonction d'un pas d'exécution en OCaml et nous l'avons placée dans une boucle "while" assurant l'exécution jusqu'à son terme.

4.4.3 La gestion des Exceptions

Représentation des exceptions

Les exceptions JavaCard sont des objets JavaCard qui sont utilisés pour manipuler et gérer les comportements hors norme de l'exécution d'un programme.

Le lancement d'exception peut être provoqué par un programme via le bytecode `athrow` ou par la machine virtuelle elle-même.

Ces exceptions peuvent sanctionner, gérer les inconsistances des données présentées au programme, des données stockées en mémoire, ou même de l'exécution. De telles inconsistances peuvent être, par exemple, un état de la machine virtuelle où une référence à un objet ne correspond à aucun objet dans le tas, le dépilement d'une pile vide, etc...

Dans le cadre de notre formalisation, les exceptions sont partitionnées en trois catégories :

1. celles qui sont lancées par le programme exécuté (via `athrow`) ;
2. les exceptions qui sont lancées par la machine virtuelle, comme l'exception provoquée par l'utilisation d'un pointeur nul ;
3. les erreurs lancées par la machine virtuelle.

La machine virtuelle peut récupérer les exceptions des deux premières classes, alors que les erreurs, dont le but du BCV est d'assurer la détection, sont fatales à l'exécution[2] en ce sens que l'erreur signale une configuration de la machine dans laquelle il est impossible de continuer l'exécution[3].

Nous fournissons le type inductif `xcpt` qui décrit les trois classes d'exceptions. Certains constructeurs prennent en paramètre une description plus détaillée de l'exception lancée. Par exemple, pour les exceptions lancées par la machine virtuelle (codée par le constructeur `JCVMException`), cette description supplémentaire est assurée par le type `xLabel`. Pour les erreurs, le descriptif supplémentaire est assuré, de façon tout à fait similaire, par le type `eLabel`.

Il est important de ne pas confondre les éléments du type `xcpt` avec les exceptions en soi, qui sont, rappelons-le, des objets. Le type `xcpt` ne fourni que la description de la nature d'une exception. Elle est à associer à la description donnée par un message d'erreur retourné par une implémentation de la machine virtuelle.

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

Inductive xcpt : Set :=
UserException :         xcpt |
JCVMException : xLabel→xcpt |
JCVMError     : eLabel→xcpt .
```

Les erreurs sont elles-mêmes divisées en trois catégories selon leur origine :

1. Les erreurs dues aux problèmes rencontrés au niveau de l'exécution du bytecode ;
2. les erreurs dues à des problèmes de gestion de la mémoire ;
3. et finalement les erreurs dues à l'interprétation des fichiers CAP.

Les erreurs de type appartiennent à la première catégorie — un exemple est l'erreur qui est lancée si la valeur qui est au sommet de la pile d'opérande n'a pas le type attendu. Une référence vers un objet qui n'existe pas produit une erreur appartenant à la deuxième classe. Un indice de méthode pointant hors de la liste de méthodes (donc vers une méthode qui n'existe pas) produira une erreur de la troisième catégorie.

Une telle classification, en dehors du gain de précision et de clarté, permet une discrimination plus aisée et un traitement plus adéquat des erreurs que le vérifieur de bytecode devra détecter [17].

[2]Rappelons que la version de la JCVM que nous avons formalisée ne supporte pas le multithreading.

[3]Il n'est pas, par exemple, possible de continuer l'exécution d'un programme si on a à dépiler dans une pile vide

Les fonctions `AbortCode`, `AbortMemory` et `AbortCap` sont utilisées, avec l'argument de type `eLabel`, pour lancer, respectivement, des erreurs de la première, deuxième et troisième catégorie.

Lancer des exceptions

Comme nous l'avons mentionné, une exception est lancée lorsqu'est exécutée l'instruction `athrow` ou quand une situation anormale est détectée par la JCVM. Dans ces deux cas, une instance de la classe de l'exception considérée est créée dans le tas, son adresse est placée au sommet de la pile d'opérande de l'environnement d'exécution courant et finalement la JCVM initie son processus de récupération d'exception.

Rattraper des exceptions

La récupération d'exception repose essentiellement sur un problème de recherche dans la pile d'exécution.

Que chercher ? Chaque méthode possède une liste de récupérateurs d'exception (se reporter à la section 4.2.3 pour plus de précision). Nous avons vu qu'un tel récupérateur spécifie l'intervalle de code dans laquelle il est capable de gérer l'exception (ou une sous-classe de l'exception) qu'il est en charge de récupérer et le code qu'il faut alors exécuter si la récupération a lieu.

Où chercher ? Ainsi, si une exception est lancée à un point donné du programme, le mécanisme de récupération vérifie s'il existe dans la méthode associée à l'environnement d'exécution courant un récupérateur adéquat. Si l'inspection du contexte d'exécution courant n'aboutit pas, alors ce contexte est dépilé et la recherche continue avec le contexte suivant.

Le processus s'arrête lorsqu'un récupérateur est trouvé ou lorsque la pile d'exécution se retrouve vide. Dans le premier cas l'exécution continue au point de programme spécifié par le récupérateur, dans le deuxième cas l'exécution est suspendue, et un état anormal (avec la description de l'erreur causée par l'exception, de type `xcpt`) est retourné. La procédure récursive qui en découle est donnée par :

```
(* Cette fonction recherche dans  la pile s un recuperateur adequat *)

Fixpoint exploreStack [s:stack] :
Class → jcprogram → (Option (bytecode_idx*stack)) :=
[cl:Class][cap:jcprogram]
Cases s of
   (* La pile est vide, donc l'exception n'est pas recuperee  *)
nil ⇒ None |
(cons el l) ⇒
   (* explore le contexte  el par l'appel de  exploreFrame    *)
   (* Si une erreur est retourne c'est qu'il n'y a pas de     *)
   (* candidat, donc la recherche continue avec le contexte   *)
   (* suivant   *)
```

```
  Cases (exploreFrame cl el cap) of
  None     ⇒ (exploreStack l cl cap)|
   (* s contient un candidat adequat, u est le point de        *)
   (* programme a partir duquel l'execution doit continuer *)
  (Some u) ⇒ (Some (u,s))
  end
end.
```

Meilleur récupérateur Précisons la notion d'adéquation. Un récupérateur h est adéquat lorsque le point de programme d'où a été lancée l'exception est contenu dans la zone où agit h et lorsque l'exception lancée est une instance de la classe (ou une instance d'une sous-classe) des exceptions que h peut prendre en compte. Nous dirons dans ce dernier cas que h couvre l'exception lancée.

Le choix d'un récupérateur adéquat se fait par l'analyse séquentielle de la liste des récupérateurs. Si un récupérateur considéré est meilleur, dans un sens que nous préciserons plus bas, que l'actuel sélectionné, alors il le remplace et la recherche se poursuit jusqu'à épuiser la liste. Ajoutons que la méthode est partielle car il peut n'y avoir aucune sélection.

La méthode ne sélectionne que des récupérateurs couvrant l'exception lancée et dont la zone d'action contient le compteur de programme. Ceci étant dit, un récupérateur h est meilleur (remplace) que le récupérateur actuellement sélectionné hs lorsque :

1. h couvre aussi l'exception lancée ;
2. le compteur de programme est aussi contenu dans la zone d'action de h
3. la zone d'action de h est strictement inclue dans la zone d'action de hs.

Remarquons que dans le cas de deux récupérateurs ayant la même zone d'action et couvrant la même exception, la préférence ira toujours au premier récupérateur inspecté.

Tous les éléments permettant la mise en place de la sémantique opérationnelle des instructions ont été présentés. Nous sommes maintenant en mesure d'exposer comment sont implémentées les instructions.

4.4.4 Sémantique des instructions

La plupart des instructions ont le même schéma d'exécution :

1. l'état initial est décomposé ;
2. des fragments de l'état sont extraits ;
3. des calculs sont effectués afin de déterminer le nouvel état ;
4. l'état final est construit sur la base des calculs et des éléments extraits.

Dans notre description, le premier point est effectué par une élimination (étude de cas, `Cases` en Coq) ; le deuxième item est constitué généralement de l'utilisation d'accesseurs de listes ; le troisième point correspond à la mise à jour de la mémoire via, éventuellement, des fonctions auxiliaires et finalement le dernier point est la détermination de l'état final (normal ou au contraire anormal).

Dans le reste de la section nous allons décrire la sémantique de cinq instructions ; `new` qui créé une nouvelle instance en mémoire, `if_acmp`, l'une des instructions de branchement conditionnel, `getfield` qui récupère la valeur d'un champ d'un objet, `invokevirtual` qui permet l'appel de méthodes virtuelles et `putstatic` qui met à jour un champ statique.

Sémantique de `new`

L'instruction `new` place dans le tas une nouvelle instance de la classe donnée en paramètre[4].

```
Definition NEW := [idx:cap_class_idx][st:dstate]
[cap:jcprogram]

let sh = (static_heap_f st) in
let hp = (heap_f st) in
let s = (stack_f st) in
Cases s  of
(cons h lf) ⇒

  (* Extrait la classe  proprietaire  du fichier CAP *)
  Cases (Nth_elt (classes cap) idx) of
  (Some cl) ⇒

    (* des lors la nouvelle instance est creee *)
    let new_obj =
      (Instance
         (Build_type_instance idx
                           (* initialisation des valeurs *)
                           (init_var (class_var cl))
                           (context_ref h)
                           false
                           false)) in

    (* et est placee dans le tas *)
    (Normal (Build_dstate
                    (* la pile statique reste inchangee *)
                    sh

                    (* le tas contient donc la nouvelle instance *)
                    ((app hp (cons new_obj (nil obj)))

                    (* son adresse est placee dans la pile d'operandes *)
                    (cons (update_opstack
                            (cons (vRef
                                     (vRef_instance idx
                                       (next_heap_idx (length hp))))
                               (opstack h)) h)
                         lf)))) |
```

[4]Par convention, les fonctions suffixées par `_f` sont les accesseurs des enregistrements considérés.

```
    None ⇒ (AbortCode class_membership_error st)
  end |
_ ⇒ (AbortCode state_error st)
end.
```

L'instance créée `new_obj` est ajoutée au tas et son adresse, calculée par `next_heap_idx` et qui correspond à `(length hp)` $+1$, est empilée dans la pile d'opérandes. À ce point, remarquons que les adresses d'objet sont des entiers naturels qui correspondent à la position de l'objet considéré dans la liste d'objets représentant le tas. Remarquons également que l'adresse d'un objet ne peut jamais être zéro du fait de l'utilisation de la valeur `(S (length hp))` pour signifier la dite position. Il est dès lors impossible que `NEW` forge le pointeur nul pour référer une nouvelle instance. Nous testons, cependant, la nullité de toute référence partout ailleurs, et pouvons donc, si nécessaire, modifier la sémantique de `NEW` de façon à ce que le pointeur nul soit retourné si une certaine quantité de mémoire a été dépassée.

Sémantique de `if_acmp`

L'instruction `if_acmp` exécute un branchement si une comparaison de référence est affirmative. Les valeurs à comparer sont dépilées de la pile d'opérande et doivent être du type `Ref`.

L'instruction `if_acmp` existe sous quatre formes dans l'ensemble des instructions de la JCVM. Les deux premières sont `if_acmpeq` et `if_acmpne` pour lesquels la comparaison réussit si, respectivement, les deux références sont égales ou différentes. Ces deux instructions sont rassemblées en une seule fonction avec un paramètre supplémentaire permettant d'établir laquelle des deux comparaisons effectuer. Il y a aussi les deux instructions `if_acmpeq_w` et `if_acmpne_w` où `w` indique qu'il s'agit de comparaison de références larges. Le JCVM Tools ayant traduit toutes ces références en entiers naturels, nous n'avons plus à faire la différence entre les références de différentes tailles. `if_acmpeq_w` et `if_acmpne_w` s'identifient donc avec `if_acmpeq` et `if_acmpne`.

Formellement nous avons :

```
Definition IF_ACMP :=
  [oper:opcmp][branch:bytecode_idx][st:dstate]

let sh = (static_heap_f st) in
let hp = (heap_f st) in
let s = (stack_f st) in
Cases s  of
(cons h lf) ⇒
  Cases (head (opstack h)) (head (tail (opstack h))) of
    (Some v2) (Some v1) ⇒
      Cases v1 v2 of
        (vRef vx) (vRef vy) ⇒
          (update_frame
            (res_pc2
                (res_acompare2 oper (vr2hi vx) (vr2hi vy))
                 branch h) st) |
        _  _ ⇒ (AbortCode type_error st)
```

```
      end|
    _ _ ⇒ (AbortCode opstack_error st)
  end |
_ ⇒ (AbortCode state_error st)
end.
```

La fonction `update_frame` prend en paramètre un contexte d'exécution h, un état s et retourne un `rdstate` normal dans lequel le contexte d'exécution en sommet de pile d'exécution (le contexte d'exécution courant) est h. Le contexte d'exécution qui est passé à `update_frame` est le contexte d'exécution courant dans lequel la pile d'opérande a été modifiée par l'application de la fonction `res_pc2`.

```
Definition res_pc2 :=
  [result:bool][branch:bytecode_idx][h:frame]
Cases result of
 true ⇒
   (update_pc
      branch
      (update_opstack (tail (tail (opstack h))) h)) |
 false ⇒
   (update_pc
      (S (p_count h))
      (update_opstack (tail (tail (opstack h))) h))
end.
```

La fonction `vr2hi` extrait la valeur numérique de la référence passée en argument. La fonction `res_acompare2` effectue la comparaison requise (l'égalité ou la différence) et le résultat est passé en paramètre à la fonction `res_pc2`. Celle-ci n'a plus qu'à modifier le pointeur de programme dans le contexte d'exécution en accord avec la valeur du test ainsi qu'à dépiler les deux valeurs comparées.

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

Sémantique de `getfield`

L'instruction `getfield` permet de consulter un champ donné d'un objet. La référence de l'objet est dépilé de la pile d'opérande de l'environnement d'exécution courant. La valeur dépilée doit donc être de type *Ref_instance* et ne doit pas être la référence nulle. Dans le cas d'une référence nulle, l'exception `NullPointerException` est lancée (voir section 4.4.3). La référence est alors utilisée pour accéder, dans le tas, à l'objet u en question. La valeur du champ désiré est retirée de la liste des champs `contents_i` de l'objet u et est alors empilée dans la pile d'opérande de l'environnement d'exécution courant. Remarquons toutefois que la valeur du champ est placée dans la pile sous la condition qu'elle appartienne bien au type attendu. Cette vérification de type est essentiellement due à la factorisation des quatre variantes de l'instruction `getfield` présentent dans l'ensemble d'instructions de la JCVM (`getfield_a`, `getfield_b`, `getfield_s` et `getfield_i`) en l'instruction `getfield`.

Ces considérations sont formalisées dans la fonction suivante où les arguments *x* et *idx* représentent respectivement la référence de l'objet et la position du champ dans la liste des champs de l'objet considéré.

```
Definition getfield_obj :=
  [ops:(list valu)][x:valu][t:type][idx:instance_field_idx]
  [st:dstate][cap:jcprogram]

(* L'etat initial est decompopse *)
let sh = (static_heap_f st) in
let hp = (heap_f st) in
let s = (stack_f st) in

  (* La reference est analysee *)
  Cases x of
  (vRef (vRef_instance  ti vx) ⇒
    (if (res_null (vRef_instance ti vx))
     then (ThrowException NullPointer st cap)
     else

       (* Extraction de l'objet reference *)
       Cases (get_inst_from_heap hp x cap) of
       (Some u) ⇒

         (* Extraction du champ *)
         Cases (Nth_elt (contents_i u) idx) of
         (Some nod) ⇒


           (* Verification des types et de la compatibilite *)
           Cases nod t of
           (vRef _)                (Ref _)
                ⇒ (res_getfield ops nod state u cap) |
           (vPrim (vByte vnod))    (Prim Byte)
                ⇒ (res_getfield  ops
                                  (vPrim (tpz2vp Short vnod))
                                  state u cap) |
           (vPrim (vBoolean vnod)) (Prim Byte)
                ⇒ (res_getfield  ops
                                  (vPrim (tpz2vp Short vnod))
                                  state u cap) |
           (vPrim (vShort _))      (Prim Short)
                ⇒ (res_getfield ops nod state u cap) |
           (vPrim (vInt _))        (Prim Int)
                ⇒ (res_getfield ops nod state u cap) |
            _                       _
                ⇒ (AbortCode type_error state)
             end |
         None ⇒ (AbortCap field_error st)
         end |
```

```
        None ⇒ (AbortMemory heap_error st)
        end
      ) |
  _ ⇒ (AbortCode type_error st)
end.
```

La fonction res_getfield qui a pour signature $(list\ valu) \rightarrow valu \rightarrow dstate \rightarrow type\_instance \rightarrow jcprogram \rightarrow frame$ permet, sous certaines circonstances, de lancer l'exception SecurityException (c'est le cas lorsque le contexte courant n'est pas le contexte propriétaire de l'objet référencé dans le tas). Dans le cas contraire cette fonction retourne l'état attendu (normal) mis à jour.
L'instruction getfield est donc formalisée de la façon suivante :

```
Definition GETFIELD :=
 [t:type][idx:instance_field_idx][st:dstate][cap:jcprogram]

(* L'etat est decompose *)
let sh = (static_heap_f st) in
let hp = (heap_f st) in
let s = (stack_f st) in
Cases s  of
(cons h lf) ⇒

  (* La reference est depilee de la pile d'operande *)
  Cases (head (opstack h)) of
    (Some x) ⇒
      (getfield_obj (tail (opstack h)) x t idx st cap) |
    None     ⇒
      (AbortCode opstack_error st)
  end |
_ ⇒ (AbortCode state_error st)
end.
```

Sémantique de invokevirtual

La sémantique de invokevirtual est légèrement plus compliquée. Cette instruction appelle une certaine méthode d'une instance de classe. Un nouvel environnement d'exécution, correspondant à l'exécution de la méthode invoquée, est créé et empilé dans la pile d'exécution. Les $nargs - 1$ arguments de la méthode et la référence à l'objet (qui est une instance de la classe de la méthode en question) sont dépilés de la pile d'opérande et sont placés dans les variables locales du nouvel environnement d'exécution. L'environnement d'exécution qui a produit l'appel est mis à jour. L'environnement créé qui est empilé dans la pile d'exécution devient alors l'environnement courant et son pointeur de programme est initialisé à 1 (sur la première instruction de la méthode). L'exécution continue ainsi avec la première instruction du nouvel environnement.

Comme pour getfield, l'exécution de cette instruction peut occasionner le lancement d'exceptions. Lorsque la référence de l'objet est nulle l'exception

`NullPointerException` est lancée. De même, si le contexte propriétaire de la méthode invoquée n'est pas le contexte actif, l'exécution de `invokevirtual` provoque le lancement de l'exception `SecurityException`. Nous avons donc :

```
Definition new_frame_invokevirtual :=
[nargs:nat][m:Method][nhp:obj][st:dstate][cap:jcprogram]

(* L'etat initial est decompose *)
let sh = (static_heap_f st) in
let hp = (heap_f st) in
let s = (stack_f st) in
Cases s  of
 nil        ⇒ (AbortCode state_error st) |
(cons h lf) ⇒


  (* Extraction de la liste des arguments *)
  Cases (l_take nargs (opstack h)) (l_drop nargs (opstack h)) of
  (Some l) (Some l') ⇒

    (* Verification de la securite *)
    (if (violation_security_invokevirtual h nhp)
     then (ThrowException Security st cap)
     else

        (* Verification de la signature *)
        (if (signature_verification (valulist_to_typelist l)
                                    (Fst (signature m)) cap)

        (* Dans le cas positif, la pile d'execution est mise a jour *)
         then (Normal
                (Build_dstate
                    sh
                    hp
                    (cons

                         (* Le nouvel environnement d'execution est *)
                         (* l'environnement courant                 *)
                         (Build_frame (nil valu)
                            (make_locvars l (local m))
                            (method_id m)
                            (get_owner_context nhp)
                            false
                            (0))

                          (* L'ancien environnement est mis a jour  *)
                          (cons (Build_frame l'
                                    (locvars h)
                                    (method_loc h)
                                    (context_ref h)
                                    (analyzed h)
                                    (p_count h))
                                 lf))))
```

```
        else (AbortCode signature_error st)
       )
    ) |
    _ _ ⇒ (AbortCode opstack_error st)
  end
end.
```

Cette fonction réalise plusieurs tests, comme `violation_security_invokevirtual`. Ce test implémente le mécanisme de pare-feu de JavaCard. Par exemple, dans le cas où l'objet `nhp` est une instance, la fonction ira vérifier si (1) le context actif est le contexte du JCRE (rappelons que le JCRE est privilégié et possède un contexte qui lui permet d'accéder à tout objet) ou (2) le contexte actif est le propriétaire de `nhp` ou finalement (3) l'instance est un point d'entrée. Dans le cas opposé, la fonction retourne `true` pour signaler l'existence d'une violation des règles de sécurité.
La fonction `new_frame_invokevirtual` est utilisée par la fonction `INVOKEVIRTUAL` définissant la sémantique de l'instruction `invokevirtual` :

```
Definition INVOKEVIRTUAL :=
[nargs:nat][nm:class_method_idx][st:dstate][cap:jcprogram]

(* L'etat initial est decompose *)
let sh = (static_heap_f st) in
let hp = (heap_f st) in
let s = (stack_f st) in

Cases s  of
 nil        ⇒ (AbortCode state_error st) |
(cons h lf) ⇒

  (* nargs, le nombre d'arguments, doit etre plus grand que zero *)
  Cases nargs of
  O     ⇒ (AbortCode args_error st) |
  (S _) ⇒

    (* Extraction de la reference d'objet (le nargs ieme element) *)
    Cases (Nth_func (opstack h) nargs) of
    None ⇒ (AbortCode opstack_error st) |
    (Some x) ⇒

      (* Teste si cet element est une reference *)
      Cases x of
      (vPrim _)  ⇒  (AbortCode type_error st) |
      (vRef vr ) ⇒

        (* extraction de l'adresse contenue dans *)
        (* vr (la valeur numerique)             *)
        let vx = (vr2hi vr) in

        (* teste si la reference est nulle *)
        (if (test_NullPointer vx)
         then (ThrowException NullPointer st cap)
         else
```

```
            (* Extraction de l'objet reference *)
            Cases (Nth_func hp vx) of
            None ⇒ (AbortMemory heap_error st) |
            (Some nhp) ⇒

              (* Cherche la classe correspondante *)
              Cases (Nth_elt (classes cap)
                             (get_obj_class_idx nhp)) of
              (Some c) ⇒

                (* Cherche la methode correspondante *)
                Cases (get_method c nm) of
                (Some m) ⇒ (new_frame_invokevirtual
                               nargs m nhp st cap) |
                None ⇒ (AbortCap methods_membership_error st)
                end |
               _ ⇒ (AbortCap class_membership_error st)
              end
            end
         )
       end
     end
   end
end.
```

L'argument nm représente l'indice de la méthode dans la classe de l'objet considéré. INVOKEVIRTUAL réalise aussi d'autres vérifications requises par la spécification de la plate-forme JavaCard. Par exemple $nargs$ ne doit pas être zéro, le $nargs^{ième}$ élément, qui représente la référence à l'objet dont on appelle la méthode, doit être du type Ref.

Sémantique de `putstatic`

Pour finir, présentons la sémantique défensive de l'instruction `putstatic`.

Il existe quatre variantes de l'instruction dans la JCVM, une pour chaque type de base (boolean, short, int et address). La factorisation de la sémantique de ces variantes en une seule fonction nous amène à considérer un paramètre supplémentaire qui est le type `t` nous permettant de considérer la sémantique de la variante correcte.

L'instruction se charge d'extraire la valeur `x` à placer dans le champ statique considéré. La valeur est celle qui est placée en sommet de la pile d'opérande du contexte d'exécution courant. Le champ est spécifié par le paramètre `idx` qui est son indice dans la liste des champs statiques. Le type de la valeur est comparé avec celui du champ. S'il y a correspondance alors les fonctions `res_putstatic` et `res_putstatic_ref` construisent un nouvel état à partir de l'ancien ou le champ considéré est actualisé. Notons qu'il faut, dans le cas d'un champ contenant une référence, vérifier qu'il n'y a pas violation de la politique de sécurité de JavaCard. `res_putstatic_ref` se démarque de `res_putstatic` précisément sur ce point.

```
Definition PUTSTATIC :=
[t:type][idx:instance_field_idx][st:dstate][cap:jcprogram]

(* L'etat initial est decompose *)
let sh = (static_heap_f st) in
let hp = (heap_f st) in
let s = (stack_f st) in
Cases s  of
nil        ⇒ (AbortCode state_error st) |
(cons h lf) ⇒

  (* La valeur au sommet de la pile d'operande est depile  *)
  Cases (pop_opstack t (opstack h)) of
  (Some (x,lv)) ⇒

   (* le champ dans lequel la valeur doit etre placee est    *)
   (* identifie et son type est compare a celui de t          *)
   Cases (Nth_elt sh idx) of
   (Some nod) ⇒
      Cases nod t of
      (vRef _)              (Ref _)     ⇒ (res_putstatic_ref
                                              lv st x idx cap) |
      (vPrim (vByte _))     (Prim Byte) ⇒ (res_putstatic
                                              lv st x idx) |
      (vPrim (vBoolean _)) (Prim Byte  ⇒ (res_putstatic
                                              lv st x idx) |
      (vPrim (vShort _))    (Prim Short)⇒ (res_putstatic
                                              lv st x idx) |
      (vPrim (vInt _))      (Prim Int)  ⇒ (res_putstatic
                                              lv st x idx) |
      _                     _           ⇒ (AbortCode type_error st)
       end |
   None ⇒ (AbortCode static_heap_error st)
   end  |
 None ⇒ (AbortCode opstack_error st)
 end
end.
```

4.5 Exécutabilité

Une des particularités de notre formalisation de la machine virtuelle est sa capacité à exécuter des programmes JavaCard. Ceci peut être fait en utilisant Coq ou en passant par le mécanisme d'extraction et par le langage OCaml. Ce point particulier nous permet la comparaison du comportement de notre formalisation avec celui des implémentations de référence (la machine virtuelle de SUN, par exemple). Il est clair qu'une telle possibilité nous permet de consolider la confiance que l'on peut avoir de la robustesse, de la fiabilité et du degré de réalisme de notre formalisation.

4.5.1 Transformation des Fichiers

Mais avant de pouvoir utiliser un programme JavaCard, le code source de ce programme doit se plier à plusieurs manipulations décrites dans la figure 2.5.

Premièrement, le code source du programme est écrit en Java mais n'utilisent que les particularités propres à JavaCard, comme l'utilisation exclusive des APIs autorisées. Celui-ci est placé dans un fichier .java puis est compilé en un fichier CLASS par un compilateur java standard. Les fichiers CLASS sont alors donnés au convertisseur de fichiers CLASS et au constructeur de fichiers CAP. Le fichier CAP résultant est la représentation finale du programme JavaCard.

En dernier lieu, afin que le fichier CAP puisse être lu au sein de notre formalisation, il faut encore le traduire en un fichier Coq. Ce processus, décrit par figure 4.1, est le propos du JCVM Tools décrit section 4.3. Ce fichier Coq peut être ensuite compilé par le compilateur Coq et finalement chargé, avec la formalisation de la machine virtuelle, dans la boucle d'interaction de Coq. Cet ensemble de définitions Coq peut ensuite, si nécessaire, être extrait vers OCaml. Comme nous l'avons déjà référée, l'exécution par OCaml est plus rapide et n'est pas limité par le critère de terminaison imposé aux fonctions Coq.

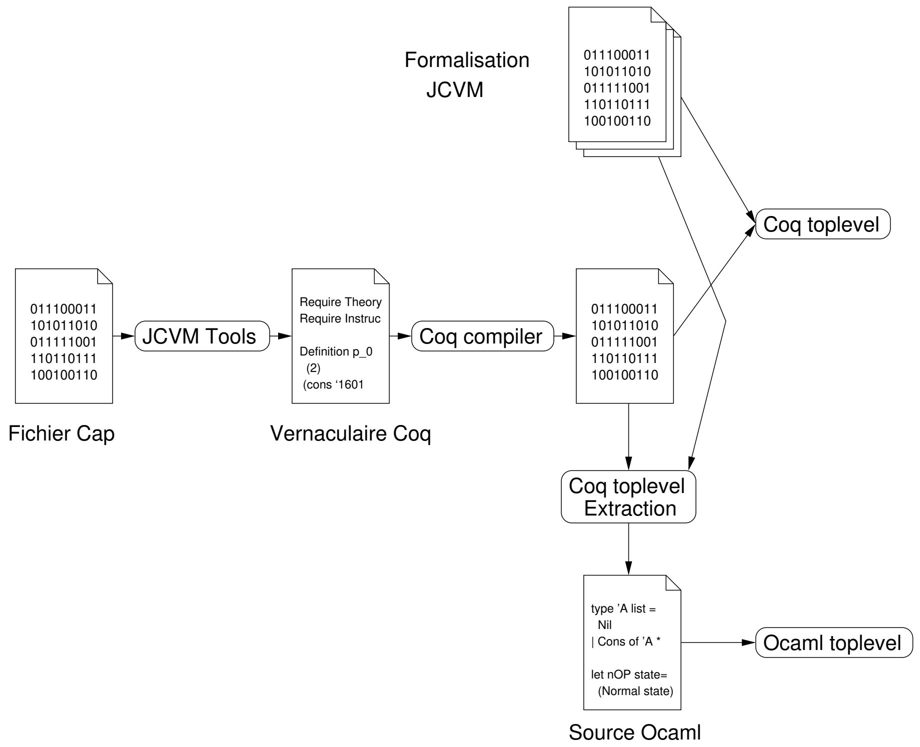

FIG. 4.1 – Transformation d'un programme Java Card.

4.5.2 Un exemple

```
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;
        }
    }
}
```

```
Definition m_18 := (Build_Method    (* min *)
  (  (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)
(18)
(13) ).
```

4.6 Une méthodologie pour l'abstraction de la machine virtuelle

Nous allons dans les sections suivantes aborder la construction de machines virtuelles qui sont des variations de la machine virtuelle défensive présentée dans les sections précédentes.

La méthodologie suivie pour obtenir de telles machines est générale et, bien que nous allons présenter une variation bien particulière, celle-ci peut être utilisée afin de produire de toutes autres machines.

Concrètement, nous aimerions pouvoir disposer d'un moyen de vérifier qu'une certaine propriété P est vérifiée par un programme p quelconque.

Un moyen de le garantir est de disposer d'une machine virtuelle qui vérifie, tout au long de l'exécution de p, qu'il n'y a pas de violation de la propriété P. Une telle machine virtuelle est qualifiée de $P-$défensive. On dit, dans ce cas, que l'on dispose d'une vérification dynamique de P.

Une autre approche est de disposer d'une vérification statique, c'est-à-dire d'une vérification qui ne nécessite pas l'exécution de p pour s'assurer de l'adhésion de p à P. Ce test est réalisé par ce qui est dénommé un $P-$vérifieur de bytecode (ou encore $P-$BCV). Celui-ci est lui même construit au dessus d'une machine qualifiée de $P-$abstraite, en ce sens qu'elle n'effectue que les tests associés à la vérification de P. Tout programme p soumis avec succès à ce BCV peut alors être exécuté sur une machine virtuelle qui n'a plus besoin de vérifier l'adhésion du programme à la propriété. Cette machine virtuelle allégée des vérifications propres à la propriété P est dite $P-$offensive. De façon informelle on peut résumer la comparaison des deux approches par la formule suivante :

$$P\!-\!défensive \;=\; P\!-\!offensive \;+\; P\!-\!abstrait \tag{4.1}$$

Cette vue globale de la vérification est tout à fait classique, et correspond bien à ce qui a été appliqué à la plate-forme JavaCard. Les spécifications de la plate-forme JavaCard fournies par SUN décrivent une machine virtuelle défensive sur les différents aspects qui forment le requis estimé minimum pour une machine virtuelle sûre. On peut ainsi citer la politique de partage d'objet, le pare-feu, le typage, l'initialisation des objets, etc. Cette description conduit naturellement à une machine virtuelle défensive. Cependant en pratique les implémentations fournissent une machine virtuelle offensive sur ces aspects et un vérifieur de bytecode qui vérifie au préalable que tout programme adhère bien aux propriétés requises.

Jusqu'à maintenant, nous avons décrit une machine virtuelle qui suit les spécifications SUN, c'est donc selon notre terminologie une machine virtuelle défensive. Vérifier une telle plate-forme n'est donc pas suffisant car sur le plan pratique ce n'est pas ce qui est implémenté. Cependant l'équation (4.1) suggère que l'on peut dériver à partir d'une propriété P et d'une machine $P-$défensive la machine $P-$offensive et le $P-$BCV. Le procédé que nous allons décrire est celui qui nous avons suivi pour réaliser et valider une telle dérivation.

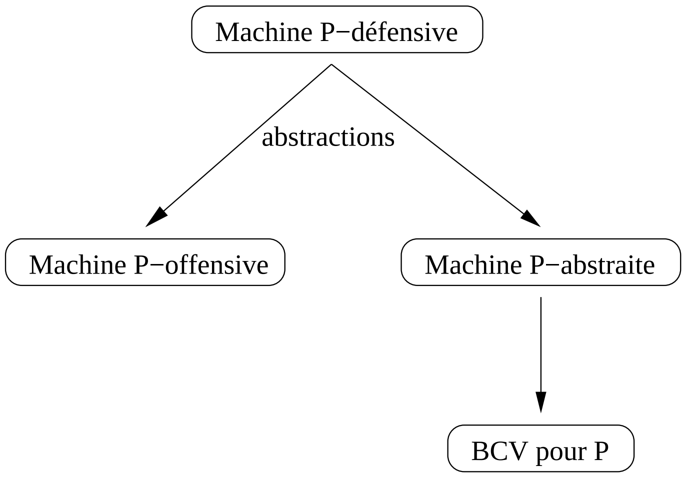

FIG. 4.2 – Méthodologie pour l'abstraction

Pour une propriété P[5], et une machine virtuelle $P-$défensive, on peut extraire une machine virtuelle $P-$offensive en transformant et adaptant la machine défensive de façon à ce que les vérifications de P disparaissent. De même un $P-$BCV peut être construit au dessus de la machine $P-$abstraite, qui est la machine défensive où toute opération autre que la vérification de P est supprimée. Traditionnellement ces deux machines sont obtenues par deux abstractions, au sens de l'abstraction abstraite, duales l'une de l'autre. On peut trouver une illustration du procédé dans la figure 4.2. La justification d'une telle construction repose alors sur la validation croisée des machines virtuelles. Cette validation consiste à prouver que (1) pour tout programme p, si la propriété P est vérifiée (le $P-$BCV valide p) alors les exécutions par la machine $P-$défensive et la machine $P-$offensive coïncident ; (2) pour tout programme p tel que l'exécution

[5] P peut être une propriété de typage, une propriété sur le flux de l'information, sur le contrôle des ressources, etc...

par la machine défensive débouche sur une violation de la propriété P alors l'exécution sur la machine $P-$abstraite débouchera sur la même violation, et le $P-$BCV ne validera pas le programme p.

Il est important de souligner quelques subtilités propres à cette approche. La validation croisée considère la validation des différentes machines les unes par rapport aux autres. Parvenir à une telle validation ne signifie pas que l'on dispose d'une plate-forme résistante à toute épreuve, mais plutôt que si l'on s'accorde sur la justesse de la machine $P-$défensive alors la justesse par rapport à P des autres machines en découle.

Même s'il n'est pas raisonnable de penser garantir formellement qu'une spécification correspond à ce que l'on voulait spécifier, il apparaît clairement comme nécessaire de compléter la validation croisée par une évaluation de la justesse de la machine défensive.

Dans ce sens, la formalisation ici présentée a été évaluée par GEMPLUS [101], les sources ont été à plusieurs occasions soumises à la communauté ainsi que rendues disponibles.

4.7 La machine virtuelle offensive

Dans cette section nous illustrons la construction d'une machine offensive (oJCVM) à partir de la machine défensive. La propriété P est ici la propriété de bon typage, c'est-à-dire qu'à tout endroit et tout moment de l'exécution toute valeur est du type espéré (sans quoi la machine défensive lance une exception dénonçant le mauvais typage). La machine offensive résultante est donc une machine virtuelle où l'on s'affranchit de tester le typage. Il est important d'insister que tout autre test se maintient. La machine n'est offensive que sur ce point.

Pour commencer, modifions notre modèle des valeurs pour qu'il corresponde à l'abstraction que l'on désire effectuer. Nous supprimons donc l'information des types :

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

Notons que `valu` est défini dans les deux versions de la machine virtuelle. Nous utiliserons donc les noms qualifiés, comme `off_JCVM` pour la machine offensive, lorsque l'exposé pourrait prêter à confusion.

Les définitions de `frame` et d'état de la mémoire (dénommé ici `ostate`) restent les mêmes mais reposent sur la nouvelle notion de valeur. Ceci conduit à un modèle simplifié de la mémoire dans lequel toute information de type est omise partout où `valu` est utilisé. Dès lors l'étape suivante consiste en la modification de la sémantique de chaque instruction. Cette modification est essentiellement une adaptation de la sémantique visant à supprimer toute vérification de typage.

Dans le reste de la section nous allons présenter la version offensive des instructions que nous avons décrite dans les sections précédentes en insistant surtout sur les adaptations sémantiques apportées.

Sémantique offensive de `new`

La version offensive de `new` diffère très peu de la version défensive. En effet aucune vérification de type particulière n'est requise par la spécification SUN. La sémantique offensive de `new` se détache alors de la sémantique défensive par le fait que l'adresse du nouvel objet placée dans la pile d'opérande est une valeur non typée.

```
Definition NEW := [idx:cap_class_idx][st:ostate][cap:jcprogram]
let sh = (static_heap_f st) in
let hp = (heap_f st) in
let s = (stack_f st) in
Cases s  of
(cons h lf) ⇒

  (* Extrait la classe  proprietaire  du fichier CAP *)
  Cases (Nth_elt (classes cap) idx) of

  (* then a new instance is created and pushed into the heap *)
  (Some cl) ⇒

    (* des lors la nouvelle instance est creee *)
    let new_obj =
         (Instance
             (Build_type_instance idx
                           (* initialisation des valeurs *)
                           (init_var (class_var cl))
                           (context_ref h)
                           false
                           false) in

   (* et est placee dans le tas *)
   (Normal (Build_ostate
                   (* la pile statique reste inchangee *)
                   sh

                   (* le tas contient donc la nouvelle instance *)
                   ((app hp (cons new_obj (nil obj)))

                   (* son adresse (non typee) est placee dans la *)
                   (* pile d'operandes                            *)
                   (cons (update_opstack
                                 (cons (inject_nat (S (length hp)))
                                    (opstack h)) h)
                      (tail s))))) |
    None ⇒ (AbortCode class_membership_error st)
  end |
_ ⇒ (AbortCode state_error st)
end.
```

Sémantique offensive de if_acmp

Pour ce qui est de la sémantique offensive de if_acmp notons tout d'abord que les versions offensive des fonctions res_acompare2 et res_pc2 sont en tout point identiques à leur version défensive.

```
Definition IF_ACMP := [oper:opcmp][branch:bytecode_idx][st:ostate]
let sh = (static_heap_f st) in
let hp = (heap_f st) in
let s = (stack_f st) in
Cases s  of
(cons h lf) ⇒
  Cases (head (opstack h)) (head (tail (opstack h))) of
  (Some vy) (Some vx) ⇒
             (update_frame (res_pc2 (res_acompare2 oper vx vy)
                                    branch h) st) |
    _          _      ⇒ (AbortCode opstack_error st)
  end  |
_ ⇒ (AbortCode state_error st)
end.
```

Remarquons donc que la vérification du type des valeurs retirées de la pile d'opérande a disparue : les valeurs retirées sont transmises telles quelles aux fonctions calculant le branchement.

Sémantique offensive de getfield

Analysons la version offensive de getfield_obj. La suppression de la vérification de type est ici encore plus drastique. La version offensive admet que le type de x est une référence à une instance, et ne teste que son égalité avec le pointeur nul. De même, le type du champs considéré n'a plus d'importance, il est correct par défaut. Ce fait rend redondant le paramètre t de la version défensive. Il est donc lui aussi supprimé. Remarquons finalement, là aussi, la disparition des branches d'erreurs débouchant sur une erreur de type.

```
Definition getfield_obj :=
[ops:(list valu)][x:valu][idx:instance_field_idx]
[st:ostate][cap:jcprogram]

(* L'etat initial est decompopse *)
let sh = (static_heap_f st) in
let hp = (heap_f st) in
let s = (stack_f st) in

  (* La reference est analysee *)
  (if (res_null x)
     then (ThrowException NullPointer st cap)
     else

       (* Extraction de l'objet reference *)
       Cases (get_inst_from_heap hp x cap) of
        (Some u) ⇒
```

```
          (*  Extraction du champ *)
            Cases (Nth_elt (contents_i u) idx) of
            (Some nod) ⇒ (res_getfield ops nod st u cap) |
            None ⇒ (AbortCap field_error st)
            end |
        None ⇒ (AbortMemory heap_error st)
        end).
```

C'est donc sans surprise que nous constatons que la version offensive de `GETFIELD` est identique à la version défensive (à l'appel de `getfield_obj` près). En effet la partie significative de la sémantique de cette instruction est capturée par la fonction `getfield_obj`

```
Definition GETFIELD :=
[t:type][idx:instance_field_idx][st:ostate][cap:jcprogram]

(* L'etat est decompose *)
let sh = (static_heap_f st) in
let hp = (heap_f st) in
let s = (stack_f st) in
Cases s  of
(cons h lf) ⇒

  (* La reference est depilee de la pile d'operande *)
  Cases (head (opstack h)) of
  (Some x) ⇒ (getfield_obj (tail (opstack h)) x idx st cap) |
  None     ⇒ (AbortCode opstack_error st)
  end |
_ ⇒ (AbortCode state_error st)
end.
```

Sémantique offensive de `invokevirtual`

La vérification du typage, dans la version défensive de `new_frame_invokevirtual`, se trouve concentrée dans la vérification de la signature que celle-ci met en place. La version offensive de la fonction `new_frame_invokevirtual` assume au contraire que tout se passe bien à ce niveau. Remarquons cependant que les vérifications de sécurité sont maintenues.

Nous avons donc :

```
Definition new_frame_invokevirtual :=
[nargs:nat][m:Method][nhp:obj][st:ostate][cap:jcprogram]

(* L'etat initial est decompose *)
let sh = (static_heap_f st) in
let hp = (heap_f st) in
let s = (stack_f st) in
Cases s  of
 nil       ⇒ (AbortCode state_error st) |
(cons h lf) ⇒


  (* Extraction de la liste des arguments *)
```

```
  Cases (l_take nargs (opstack h)) (l_drop nargs (opstack h)) of
  (Some l) (Some l') ⇒

     (* Verification de la securite *)
    (if (violation_security_invokevirtual h nhp)
     then(ThrowException Security st cap)
     else
        (Normal
          (Build_ostate
             sh
             hp
            (cons

                (* Le nouvel environnement d'execution est *)
                (* l'environnement courant                 *)
                (Build_frame (nil valu)
                             (make_locvars l (local m))
                             (method_id m)
                             (get_owner_context nhp)
                             (0))

                              (* L'ancien environnement est mis a jour *)
                                     (cons (Build_frame l'
                                                       (locvars h)
                                                       (method_loc h)
                                                       (context_ref h)
                                                       (p_count h))
                                     lf ))))) |
  _ _ ⇒ (AbortCode opstack_error st)
  end
end.
```

La contrepartie offensive de INVOKEVIRTUAL suit les mêmes principes. La valeur extraite de la pile d'opérande est forcément une référence à un objet, il n'est donc pas nécessaire de vérifier son type.

```
Definition INVOKEVIRTUAL :=
[nargs:nat][nm:class_method_idx][st:ostate][cap:jcprogram]

(* L'etat initial est decompose *)
let sh = (static_heap_f st) in
let hp = (heap_f st) in
let s = (stack_f st) in
Cases s  of
 nil       ⇒ (AbortCode state_error st) |
(cons h lf) ⇒

  (* nargs, le nombre d'arguments, doit etre plus grand que zero *)
  Cases nargs of
  O     ⇒ (AbortCode args_error st) |
  (S _) ⇒

    (* Extraction de la reference d'objet (le nargs ieme element) *)
    Cases (Nth_func (opstack h) nargs) of
```

```
    None ⇒ (AbortCode opstack_error st) |
    (Some vx) ⇒
        (* teste si la reference est nulle *)
        (if (test_NullPointer vx)
         then (ThrowException NullPointer st cap)
         else

            (* Extraction de l'objet reference *)
            Cases (Nth_func hp (absolu vx)) of
            None ⇒ (AbortMemory heap_error st) |
            (Some nhp) ⇒

              (* Cherche la classe correspondante *)
              Cases (Nth_elt (classes cap) (get_obj_class_idx nhp)) of
              (Some c) ⇒

               (* Cherche la methode correspondante *)
                Cases (get_method c nm cap) of
                (Some m) ⇒ (new_frame_invokevirtual nargs m nhp st cap) |
                None ⇒ (AbortCap methods_membership_error st)
                end |
               _ ⇒ (AbortCap class_membership_error st)
              end
            end
        )
      end
  end
end.
```

Sémantique offensive de `putstatic`

Comme dernier exemple, nous introduisons la sémantique offensive de PUTSTATIC que nous utiliserons pour illustrer l'étape de validation croisée qui suit.

```
Definition PUTSTATIC :=
[t:type][idx:static_heap_idx][st:ostate][cap:jcprogram]

Cases (get_topframe st) of
(Some h) ⇒
  Cases (head (opstack h)) of
  (Some x) ⇒
    Cases (Nth_elt (get_static_heap st) idx) of
    (Some nod) ⇒
      Cases t of
        (Ref _)  ⇒ (res_putstatic_ref st x idx cap) |
        (Prim _) ⇒ (res_putstatic st x idx)
      end |
    None ⇒ (AbortCode static_heap_error st)
    end |
  None ⇒ (AbortCode opstack_error st)
  end |
None ⇒ (AbortCode state_error st)
```

```
end.
```

Il y a plusieurs instructions PUTSTATIC dans l'ensemble d'instructions de JavaCard. Ainsi, le test des différents cas possible de valeur du paramètre t n'est pas une vérification de type. Ce paramètre est uniquement utilisé ici pour permettre la factorisation de ces instructions en une seule fonction et continue donc d'être pris en compte dans la version offensive. L'instruction PUTSTATIC est chargée de placer dans un champ statique une valeur du type adéquat (du type du champ). Ces vérifications sont omises, et la version offensive se charge de retrouver le champ dans lequel placer la valeur sans en vérifier le type.

4.7.1 Équivalence des machines offensive et défensive

Nous avons vu que le procédé de construction de la machine offensive est inspiré de l'interprétation abstraite. De fait, il est possible de définir une fonction `alpha_do_valu` de type `valu` → `off_JCVM.valu` qui fait correspondre à chaque valeur de la machine virtuelle défensive une valeur de la machine virtuelle offensive par la suppression de l'information du type :

```
Definition alpha_do_valu_prim [v:valu_prim] : off_JCVM.valu_prim :=
Cases v of
vReturnAddress x ⇒ (inject_nat x)  |
vBoolean  x      ⇒ x               |
vByte x          ⇒ x               |
vShort x         ⇒ x               |
vInt x           ⇒ x               |
vInt2 x          ⇒ x
end.

(* ... *)

Definition alpha_do_valu [v:valu] : off_JCVM.valu :=
Cases v of
vPrim x  ⇒ alpha_do_valu_prim x |
vRef x ⇒ alpha_do_valu_ref x
end.
```

où `alpha_do_valu_prim` (respectivement `alpha_do_valu_ref`) est la fonction d'abstraction des valeurs du type `valu_prim` (respectivement `valu_ref`).

La fonction `alpha_do_valu` est ensuite étendue aux contextes d'exécution :

```
Definition alpha_do_frame [h:frame] : off_JCVM.frame :=
  (off_JCVM.Build_frame (map alpha_do_valu (opstack h))
                        (map alpha_do_valu (locvars h))
                        (method_loc h)
                        (context_ref h)
                        (p_count h)).
```

De manière similaire `alpha_do` est étendue à la représentation des états et des états de retour.

Pour finir, la sémantique de chaque instruction est définie de façon à ce que le diagramme de la figure 4.3 commute sous l'hypothèse que la machine virtuelle ne lance pas d'erreur de type (c'est-à-dire que la JCVM défensive n'appelle pas la fonction `AbortCode` avec le paramètre `type_error` ou tout autre label signifiant une erreur de type). Remarquons que l'établissement de cette hypothèse nous éloigne légèrement du cadre classique de l'interprétation abstraite où la commutation se formule traditionnellement inconditionnellement.

$$\begin{tikzcd}
\texttt{dstate} \arrow[r, "\texttt{dexec_instruction i}"] \arrow[d, "\texttt{alpha_do}"] & \texttt{rdstate} \arrow[d, "\texttt{alpha_do_rs}"] \\
\texttt{ostate} \arrow[r, "\texttt{oexec_instruction i}"] & \texttt{rostate}
\end{tikzcd}$$

FIG. 4.3 – Diagramme commutatif de l'exécution des machines défensive et offensive

Nous présentons dans ce qui suit une preuve formelle du diagramme en Coq qui a été développé par G. Dufay. L'équivalence entre les machines offensive et défensive est formulée comme une propriété de commutation entre les deux exécutions, sous l'hypothèse que l'exécution sur la machine virtuelle défensive ne provoque pas d'erreur de type. Donc, formellement nous voulons prouver que :

```
(∀st:dstate)((dexecs n p) ≠ (AbortCode type_error st))) →
   (alpha_do_rs (dexecs n p)) = (oexecs n (alpha_do p))
```

où `dexecs n p` (resp. `oexecs`) représente le résultat de `n` pas d'exécution de la machine défensive (resp. offensive) à partir de l'état initial associé au programme `p`.

Nous avons supposé, par soucis de clarté, que toute erreur de type a pour label `type_error`, ce qui n'est pas tout à fait vrai dans notre formalisation, comme en témoigne le label `signature_error`.

Une telle preuve, globale, est réalisée en Coq en prouvant la propriété de commutation pour chaque instruction. Pour `putstatic`, ceci nous conduit à la preuve de l'énoncé suivant :

```
Lemma PUTSTATIC_commut : (st:dstate)(t:type)(n:nat)(cap:jcprogram)
  let res = (dexec_instruction (putstatic t n) st cap) in
  let off_res =
    (oexec_instruction (putstatic t n) (alpha_do st) cap) in
  (~res = (AbortCode type_error st)) →
      (alpha_do_rs res cap) = off_res.
```

La preuve exploite un module Coq développé par P.Courtieu [16] qui gère les principes d'inversion de fonctions. Nous retrouverons ce module section 5.4. En résumé, la tactique définie dans ce module permet de remplacer une preuve de $\forall \vec{x}.\ \phi(\vec{x}, f\vec{x})$, où f est une fonction récursive définie par une analyse de cas arbitrairement complexe, par la preuve, plus petite et plus simple à mettre en œuvre, de propriétés de la forme :

$$\forall \vec{x}.\ s_1 = t_1 \to\ \ldots\ \to s_n = t_n \to \phi(\vec{x}, f\vec{x})$$

où les égalités correspondent à un cas possible dans la définition de la fonction, et chaque sous-but généré correspond à un cas possible.

Cette tactique est très utile à la preuve de propriété de commutation puisque les égalités qu'elle génère rendent, pour la plupart, la preuve de $\phi(\vec{x}, f\vec{x})$ triviale.

La taille des scripts de preuve s'en retrouve largement diminuée par rapport aux preuves sans utilisation de cette tactique.

En utilisant cette tactique, sous le contexte `st:dstate`, `t:type`, `n:nat`, `cap:jcprogram`, et si `G` est le terme correspondant au lemme `PUTSTATIC_commut` sans la présence des quantificateurs universels nous obtiendrons 40 sous-but à prouver, tels que :

- Sous-but 1 :
  ```
  (sh:static_heap)(hp:heap) st=(Build_dstate sh hp (nil frame)) →G
  ```
- Sous-but 2 :
  ```
  (sh:static_heap)(hp:heap)(f:frame)(lf:(list frame))
  st=(Build_dstate sh hp (cons h lf)) →(head (opstack h))=None →G
  ```
- Sous-but 8 :
  ```
  (sh:static_heap)(hp:heap)(f:frame)(lf:(list frame))(x:valu)(z:Z)
  st=(Build_dstate sh hp (cons h lf)) →(head (opstack h))=(Some x) →
  (Nth_elt (get_static_heap st) idx)=(Some ((Prim Short), z)) →
  t=(Prim Int) →G
  ```
- Sous-but 9 :
  ```
  (sh:static_heap)(hp:heap)(f:frame)(lf:(list frame))(x:valu)(z:Z)
  st=(Build_dstate sh hp (cons h lf)) →(head (opstack h))=(Some x) →
  (Nth_elt (get_static_heap st) idx)=(Some ((Prim Int), z)) →
  t=(Prim Int) →G
  ```

Les égalités générées sont automatiquement récrites dans `G` conduisant à des preuves triviales.

Pour le sous-but 9, qui décrit la propriété à prouver pour une configuration dans laquelle l'exécution se comporte normalement, avec des types `Int` (`res_putstatic` a été appelé et sa définition a été dépliée), `G` est récrit en :

```
  ~(Normal ((l_update_nth sh idx x),
            (hp,
            (cons (update_opstack (tail opstack h) h)
              (tail (cons h lf))))))
  =(off_JCVM.AbortCode type_error (Build_dstate sh hp (cons h lf)))
→(alpha_do_rs (Normal
                  ((l_update_nth sh idx x),
                  (hp,
                  (cons (update_opstack (tail opstack h) h)
                    (tail (cons h lf))))))
    cap)
 =(off_JCVM.Normal
      ((l_update_nth (alpha_do_sh sh) idx (alpha_do_valu x)),
       ((alpha_do_heap hp),
        (cons (off_JCVM.update_opstack
                        (lvalu2lovalu (tail (opstack h)))
                        (alpha_do_frame h))
          (tail (cons (alpha_do_frame h)
                   (alpha_do_lframe lf)))))))
```

Les hypothèses sont inutiles ici car, à une étape de réécriture près, l'égalité identifie ici deux éléments effectivement égaux.

4.8 La machine virtuelle abstraite

Dans cette section nous abordons la construction d'une machine virtuelle abstraite (aJCVM) par rapport à la propriété de bon typage.

Elle sert de base à la construction du BCV menée à terme par G.Dufay[17, 59].

Principe sous-jacent Par opposition à la machine offensive, cette machine ne travaille qu'avec les types en oubliant donc les valeurs. Elle mène donc l'exécution de façon à n'effectuer que des vérifications et des calculs de type.

Mise en place Donc, pour commencer, nous devons supprimer la composante valeur numérique des données `valu`. Cependant, toute valeur numérique ne peut être supprimée sans un minimum de précaution. En effet la présence de la valeur numérique associée au type `ReturnAddress` est impérative à l'exécution, même dans la machine abstraite. La valeur numérique de type `ReturnAddress` est utilisée pour brancher l'exécution sur l'adresse (sur l'instruction) que cette valeur représente. Les types `valu_prim` `valu_ref` et `valu` deviennent donc :

```
Inductive valu_prim : Set :=
          tReturnAddress : bytecode_idx → valu_prim |
          tVoid          : valu_prim |
          tBoolean       : valu_prim |
          tByte          : valu_prim |
          tShort         : valu_prim |
          tInt           : valu_prim |
          tInt2          : valu_prim.


Mutual Inductive valu : Set :=
       tPrim : valu_prim → valu |
       tRef : valu_ref → valu
with valu_ref : Set :=
       tRef_null : valu_ref |
       tRef_array : valu → valu_ref |
       tRef_instance : cap_class_idx → valu_ref |
       tRef_interface : cap_interf_idx → valu_ref.
```

Dès lors qu'il n'y a plus de valeur numérique, les calculs numériques sur les éléments de `valu` sont supprimés de la sémantique des instructions. Par exemple l'instruction `add` ne fait plus que vérifier si la pile d'opérandes contient deux types compatibles (pour l'addition) à son sommet, et les remplace par le type de retour adéquat. L'opération d'addition en soi n'a plus de sens et n'existe donc plus.

En poussant jusqu'au bout les conséquences de la disparition des valeurs, nous constatons qu'il n'est plus aussi nécessaire de conserver la pile de contextes d'exécution, le tas statique et le tas.

Pour ce qui est du tas, puisqu'il n'y a plus de valeurs, seul le type de l'objet devient pertinent. Le tas peut donc cesser d'être considéré dans la machine virtuelle abstraite. Créer un objet revient à pousser son type dans la pile d'opérande.

Si on continue le processus d'abstraction des valeurs numériques, le tas statique devient une liste de type. Une fois déterminé, le tas est donc *constant* puisque le type statique ne change pas au cours de l'exécution. Or c'est exactement cette information qui est contenue dans le champ `static_heap_type` de la représentation du programme considéré. Afin d'éviter une redondance d'information, le tas statique a été retiré de la représentation de l'état de la machine.

Notion d'exécution abstraite La pile de contextes d'exécution peut laisser place au contexte d'exécution courant. En effet l'exécution n'a plus besoin d'empiler des contextes d'exécution pour produire le résultat escompté. On dit, dans ce cas précis, qu'on a une *exécution méthode par méthode*. Le choix de l'exécution méthode par méthode se justifie par les deux points suivants :

- La notion de pile d'exécution n'a plus de sens lorsque ne sont manipulés que des types. En effet seul le type de retour est utile lorsqu'on considère un appel de méthode puisque cette dernière n'est appelée que pour effectuer un calcul dont la valeur appartient à son type de retour. Or celui-ci est connu, il suffit donc de placer le type dans la pile d'opérande et de passer à l'instruction suivante.
- De façon plus pragmatique, un BCV basé sur une spécification à exécution méthode par méthode est plus simple à définir et plus efficace.

L'état de la machine virtuelle abstraite se résume donc à un contexte d'exécution abstrait :

```
Definition astate := frame.
```

Introduction du non déterminisme Quelques instructions, par exemple les instructions de branchement conditionnel reposant sur la comparaison de valeurs (tel que `ifnull` ou `tableswitch`), ont la particularité de conduire à des points de programme différents selon les valeurs placées dans la pile d'opérande. Puisque ces valeurs sont perdues ces instructions, dans leur version abstraite, deviennent non-déterministes et leur sémantique doit considérer tous les cas possibles. Ainsi celles-ci retournent tous les états de retour possibles (c'est-à-dire un état pour chaque valeur possible du pointeur de programme). Ces états sont regroupés dans une liste.

Comme pour la machine offensive, nous allons, dans le reste de la section, présenter les adaptations sémantiques apportées pour obtenir la version abstraite des instructions que nous avons décrites dans les sections précédentes.

Sémantique abstraite de `new`

La sémantique abstraite de `new` se retrouve très simplifiée. Le tas n'étant plus nécessaire, la création d'un objet ne consiste, sous la condition que la classe de l'objet existe, qu'au placement de la valeur référence dans la pile d'opérande du contexte d'exécution courant. C'est `update_frame` qui effectue cette opération.

Rappelons que dans ce cas les valeurs sont des types, la valeur en question est donc le type de la référence de l'objet.

```
Definition NEW :=
[idx:cap_class_idx][h:astate][cap:jcprogram]


  Cases (Nth_elt (classes cap) idx) of
    (Some cl) ⇒ (update_frame (update_opstack
                                   (cons (tRef (tRef_instance idx))
                                         (opstack h)) h) h) |
     None ⇒ (AbortCode class_membership_error h)
   end.
```

Sémantique abstraite de `if_acmp`

L'instruction `if_acmp` est l'une des instructions introduisant un comportement non-déterministe. Celui-ci se caractérise par le fait que le test (booléen) permettant de savoir vers quelle instruction l'exécution doit se diriger, n'est plus en mesure d'être réalisé. Pour capturer un tel comportement, on veut que les deux exécutions possibles soient calculées (celle qui découle d'un test vrai et celle qui découle d'un test faux). Comment procéder ?

La solution adoptée consiste à remplacer le paramètre de `if_acmp` indiquant le test à effectuer par le résultat (donc prédéfini) du test. La fonction `exec_instr` devra alors se charger d'appeler `if_acmp` deux fois (une pour le test vrai et l'autre pour le test faux) et de collecter les deux états de retour dans une liste. Ceci donne :

```
Definition exec_instr  :=  [i:Instruction][h:astate][cap:jcprogram]
Cases i of
   ...

   (if_acmp ope branch) ⇒
           (cons (IF_ACMP true branch h)
           (cons (IF_ACMP false branch h) (nil ?))) |
   ...
```

La sémantique abstraite de `if_acmp` consiste alors à retirer les deux types de la pile d'opérande du contexte d'exécution courant, de vérifier que ce sont des références et d'actualiser le contexte d'exécution en fonction de la valeur du paramètre booléen `b`. La fonction `res_pc2` est inchangée. Comme on le voit, il n'y a plus de calcul, la valeur est directement fournie par `exec_instr`.

```
Definition IF_ACMP :=  [b:bool][branch:bytecode_idx][h:astate]

Cases (head (opstack h)) of
(Some v2) ⇒
  Cases (head (tail (opstack h))) of
  (Some v1) ⇒
    Cases v1 v2 of
     (tRef _) (tRef _) ⇒ (update_frame (res_pc2 b branch h) h) |
     _        _        ⇒ (AbortCode type_error h)
```

```
    end |
  None ⇒ (AbortCode opstack_error  h)
  end |
None ⇒ (AbortCode opstack_error h)
end.
```

Sémantique abstraite de getfield

Les transformations qui ont mené à la sémantique abstraite de `getfield` ne sont, quant à elles, plus aussi évidentes. Comment accéder à un champ d'objet lorsqu'il n'y a plus d'objet ? Dans la sémantique abstraite, la question se reformule en terme de types. Quel est le type du champ auquel on veut accéder ? La classe de l'objet possède cette information. Le type étant connu, il est comparé au type attendu, à l'image de la version défensive. Dans le cas positif, le type du champ est poussé dans la pile d'opérande. Remarquons qu'il n'y a plus de configuration conduisant au lancement de l'exception `NullPointer` ni de l'exception `SecurityException`. La fonction `t2tt` permet la conversion des éléments de `type` vers `valu`.

```
Definition GETFIELD :=
[t:type][idx:instance_field_idx][h:astate][cap:jcprogram]


(* extraction de la reference de l'objet dont on veut le champ *)
Cases (head (opstack h)) of
(Some x) ⇒

 let ops = (tail (opstack h)) in
 (* La reference est analysee *)
  Cases x of
  (Ref vr) ⇒

     (* Si la reference n'est pas une reference d'instance  *)
     (* la classe choisie est la classe java.lang.Object         *)
     let cidx = extract_class_idx vr in

      (* Le type du champ est extrait via la classe de l'objet *)
      Cases (Nth_elt (classes cap) cidx) of
      (Some cl) ⇒
          Cases (Nth_elt (class_var cl) idx) of
          (Some nod) ⇒

              (* Le type du champ est compare au type attendu  *)
              Cases nod  t of
              (Ref _)        (Ref _)      ⇒
                       (update_frame
                         (push_opstack (t2tt nod) ops h) h) |
              (Prim tByte)    (Prim Byte)  ⇒
                       (update_frame
                         (push_opstack (tPrim tShort) ops h) h) |
              (Prim tBoolean) (Prim Byte)  ⇒
                       (update_frame
```

```
                          (push_opstack (tPrim tShort) ops h) h) |
                 (Prim tShort)    (Prim Short) ⇒
                          (update_frame
                            (push_opstack (t2tt nod) ops h) h)   |
                 (Prim tInt)      (Prim Int)   ⇒
                          (update_frame
                            (push_opstack (t2tt nod) ops h) h) |
                 _                _            ⇒
                          (AbortCode type_error h)
                 end |
             None ⇒ (AbortCode field_error h)
             end |
         None ⇒ (AbortCap class_membership_error h)
         end |
     _ ⇒ (AbortCode type_error h)
     end
 None ⇒ (AbortCode opstack_error h)
 end.
```

Sémantique abstraite de invokevirtual

La sémantique abstraite de l'instruction invokevirtual est celle qui pose les problèmes les plus subtils. Il y a principalement deux points problématiques à résoudre. Premièrement, la perte des valeurs numériques nous a amené à perdre les objets. Or c'est à partir de ceux-ci que l'on obtenait la classe où chercher la méthode appelée. Deuxièmement, la perte de la pile d'exécution a une incidence particulière sur l'appel de méthode, puisqu'il n'y a plus d'empilement de contextes d'exécution.

Extraction de la classe Dans sa version défensive (et offensive) la méthode appelée est retrouvée en scrutant dans la classe de l'objet extrait. Dans le cas abstrait, il n'est plus possible d'avoir la classe dynamique. La classe dynamique, dans la version défensive, est obtenue à partir de l'objet nhp par (get_obj_class_idx nhp) (ligne 44 de la sémantique défensive de invokevirtual. Pour plus de précisions se reporter à la section 4.4.4). C'est donc la classe statique qui est choisie. Celle-ci est retrouvée à partir de la référence de l'objet r par (get_type_ref_class_idx r). La classe obtenue n'est pas forcément la classe obtenue par la machine défensive. La seule garantie que l'on ait est que la classe de la version abstraite soit une super-classe de la classe dynamique qui définit la méthode invoquée. C'est exactement ce point qui introduit les problèmes de sous-typage que l'on rencontrera dans la preuve de validation croisée de la machine abstraite (se référer à la section 4.8.1). Lorsque la méthode m est extraite, le passage de paramètres donne lieu à une vérification du typage de ceux-ci par rapport à la signature de m.

Appel et retour de méthode Lorsque l'extraction de tous les éléments nécessaires à l'appel a eu lieu, et que la vérification de ceux-ci est positive, cet appel de méthode consiste à extraire le type du retour tr de la méthode, de débarrasser la pile d'opérande des paramètres de `m` et d'empiler `tr` dans cette dernière. L'exécution devra alors continuer sur la prochaine instruction.

On obtient ainsi la fonction `INVOKEVIRTUAL` suivante :

```
Definition INVOKEVIRTUAL :=
[nargs:nat][nm:class_method_idx][h:astate][cap:jcprogram]


(* nargs doit etre plus grand que zero *)
  Cases nargs of
  O     ⇒ (AbortCode args_error h) |
  (S _) ⇒

    (* Extraction de la reference de l'objet (le nargs ieme element) *)
    Cases (Nth_func (opstack h) nargs) of
    None ⇒ (AbortCode opstack_error h) |
    (Some x) ⇒

      (* teste si cet element est une reference *)
      Cases x of
      (tRef r) ⇒

        (* extrait la classe *statique*  correspondante *)
        Cases (Nth_elt (classes cap) (get_type_ref_class_idx r)) of

        (Some c) ⇒
          Cases (get_method c nm cap) of
          (Some m) ⇒

             (* Extraction des parametres *)
             Cases (l_take nargs (opstack h)) (l_drop nargs (opstack h)) of
             (Some l) (Some l') ⇒

                (* Verification par rapport a la signature *)
                (if (signature_verification (valulist_to_typelist l)
                                            (Fst (signature m)) cap)

                (* en cas de succes, le contexte d'execution est actualise *)
                 then
                   (Normal (Build_astate
                                 (Build_frame

                                   (* le type de retour de m est empile *)
                                   (app_return_type l' (Snd (signature m)))
                                   (locvars h)
                                   (method_loc h)
                                   (S (p_count h)))))
                 else (AbortCode signature_error h)) |
             _ _ ⇒ (AbortCode opstack_error h)
            end  |
          None ⇒ (AbortCap methods_membership_error h)
          end |
        _ ⇒ (AbortCap class_membership_error h)
```

```
        end |
      _ ⇒ (AbortCode type_error h)
      end
    end
  end.
```

Sémantique abstraite de `putstatic`

Pour finir, voici la version abstraite de `putstatic`. Celle-ci se contente de dépiler la valeur (type) `x` de la pile d'opérande puis d'extraire du tas statique le type désigné par le paramètre `idx`. À l'image de la version défensive, celui-ci est comparé avec le type attendu (le type `t`). Finalement, dans le cas d'une comparaison positive, `x` remplace la valeur du champ statique.

```
Definition PUTSTATIC :=
 [t:type][idx:instance_field_idx][h:astate][cap:jcprogram]

let sh = (sheap_type_f cap) in

Cases (pop_opstack t (opstack h)) of
(Some (x,lv)) ⇒
  Cases (Nth_elt sh idx) of
  (Some nod) ⇒
     Cases nod  t of
      (Ref _)        (Ref _)       ⇒
                   (Normal (update_opstack lv h)) |
      (Prim Byte)     (Prim Byte)   ⇒
                   (Normal ((update_opstack lv h)) |
      (Prim Boolean) (Prim Byte)   ⇒
                   (Normal (update_opstack lv h)) |
      (Prim Short)   (Prim Short)  ⇒
                   (Normal (update_opstack lv h)) |
      (Prim Int)     (Prim Int)    ⇒
                   (Normal (update_opstack lv h)) |
     _               _            ⇒ (AbortCode type_error h)
     end |
  None ⇒ (AbortCode sheap_error h)
  end |
None ⇒ (AbortCode opstack_error h)
end.
```

4.8.1 Équivalence des machines abstraite et défensive

Fonctions d'abstraction À l'image de l'abstraction offensive, la construction de la aJCVM peut être formulée en terme d'abstraction. Concrètement, la notion de valeur abstraite `abs_JCVM.valu` peut être dérivée de `valu` par la fonction `alpha_da_valu: valu →abs_JCVM.valu`, et les valeurs de `valu_prim` en `abs_JCVM.valu_prim` par la fonction `alpha_da_valu_prim: valu_prim →abs_JCVM.valu_prim`.

Ces modifications sont ensuite répercutées à toute la structure de données de la mémoire de la aJCVM. Donnons en exemple les fonctions `alpha_da_valu_prim` et `alpha_da_state` :

```
Definition alpha_da_valu_prim : valu_prim → abs_JCVM.valu_prim :=
[v:valu_prim]
  Cases v of
    vReturnAddress n ⇒ (tReturnAddress n) |
    vVoid n          ⇒ tVoid |
    vBoolean n       ⇒ tBoolean |
    vByte n          ⇒ tByte |
    vShort n         ⇒ tShort |
    vInt n           ⇒ tInt |
    vInt2 n          ⇒ tInt2
  end.

Definition alpha_da_state := [s:dstate][cap:jcprogram]
  Cases (stack_f s) of
    (cons h lf) ⇒ (alpha_da_frame h) |
    nil         ⇒ default_aframe
  end.
```

La fonction `alpha_da_frame: frame →abs_JCVM.frame` est la fonction qui abstrait les contextes d'exécution et la constante `default_aframe:abs_JCVM.frame`, puisque la fonction d'abstraction doit être totale sur `astate`, est le contexte d'exécution considéré par défaut.

Propriété de correction Une fois la machine virtuelle abstraite définie, nous devons établir la correction de celle-ci par rapport à la machine défensive. Dans le cadre très particulier de l'abstraction par suppression des valeurs, la propriété de correction prend la forme de la preuve du respect d'un invariant (consulter [17]). Pour les instructions dont l'exécution reste dans la frame courante — les instructions *intra-procédurales*, le respect de l'invariant s'exprime en terme de commutation de diagramme (figure 4.4). La démonstration pour les autres instructions, mise en œuvre par Guillaume Dufay [17] dans le module BCV, est beaucoup plus complexe. Mentionnons simplement qu'elle ne prend pas la forme d'une propriété de commutation.

$$\begin{tikzcd}
\texttt{dstate} \arrow[r, "\texttt{dexec_instruction i}"] \arrow[d, "\texttt{alpha_da}"] & \texttt{rdstate} \arrow[d, "\preceq\texttt{alpha_da_rs}"] \\
\texttt{astate} \arrow[r, "\texttt{aexec_instruction i}"] & \texttt{(list rastate)}
\end{tikzcd}$$

FIG. 4.4 – Diagramme de commutation de l'exécution des machines défensive et abstraite pour les instructions intra-procédurales

La commutation s'exprime, dans le cas de la machine abstraite, par le fait que l'abstraction de l'état de retour concret est, modulo sous-typage, un élément de la liste des états de retour abstraits. Rappelons que l'existence de la liste est due au non-déterminisme de la machine abstraite. C'est cette définition de la commutation que représente la flèche de droite du diagramme.

Donc formellement et dans le cas général, nous avons à prouver :

$$
\begin{array}{l}
(\forall\ \texttt{p:jcprogram})(\forall\ \texttt{st:dstate})\ \texttt{(Hypothèses st p)} \rightarrow \\
\qquad \texttt{(alpha\_da\_rs (dexecs n st p))} \in_{\preceq} \\
\qquad \texttt{(aexecs n (alpha\_da\_state st) p)}
\end{array}
$$

où `dexecs n p` (resp. `aexecs`) représente le résultat de `n` pas d'exécution de la machine défensive (resp. abstraite) à partir de l'état initial associé au programme `p`. Le symbole $\in_{\preceq}$ représente l'appartenance modulo sous-typage. La condition initiale `(Hypothèses st p)` formule l'hypothèse que nous ne considérons que les exécutions défensives qui ne débouchent pas sur un lancement d'exception autre que les exceptions (erreurs) lancées par l'appel de `AbortCode`. Cette hypothèse est nécessaire car le seul traitement d'erreur qui a guidé la construction de la machine abstraite est le traitement des erreurs lancées par la fonction `AbortCode`. Les autres, comme les exceptions lancées par le programme (symbolisées par l'appel de la fonction `ThrowException`) sont l'apanage des machines défensive et offensive. Dans `(Hypothèses st p)` nous incluons aussi les considérations propres au changement de représentations de la mémoire entre mémoire défensive et mémoire abstraite comme la correspondance entre le tas statique défensif et le tas statique abstrait.

Procédé de démonstration Le procédé de démonstration, bien que plus complexe, suit les mêmes lignes et techniques que les démonstrations de commutation de la machine offensive. C'est pourquoi nous ne soulignons ici que les différences majeures.

Seules les instructions de branchement utilisent l'appartenance, la liste considérée pour les autres instructions sont des listes à un élément. De plus, pour la plupart des bytecodes, on peut considérer directement l'appartenance $\in$, au lieu de $\in_{\preceq}$. Cette dernière n'est utilisée que dans les instructions manipulant le tas (les instructions de manipulation d'objet comme `getfield` ou encore `putfield`). Nous illustrons l'usage de $\in_{\preceq}$ avec l'instruction `invokevirtual`, au paragraphe suivant. Pour les autres instructions impliquées, son usage est dû aux mêmes causes : recherche d'information sur la base du type statique, plutôt que sur son type dynamique, de l'information considérée. La démonstration de commutation suit ainsi le même procédé.

Un exemple d'énoncé Illustrons maintenant les ajustements générés par l'absence de pile d'exécution (de type `stack`, rappelons le). Ceux-ci consistent à ne considérer que les exécutions défensives sur des piles contenant explicitement au moins un contexte d'exécution. Le cas où l'on a à considérer le contexte d'exécution par défaut (ligne 4 de la fonction `alpha_da_state`) n'est donc jamais pris en compte. Ceci conduit, pour le cas de l'instruction `putstatic`, à l'énoncé suivant :

```
Lemma PUTSTATIC_commut : (cap:jcprogram)(sh:static_heap)
    (hp:heap)(h:frame)(lf:(list frame))(t:type)(n:nat)
  let st=(Build_dstate sh hp (cons h lf)) in
  let res=(dexec_instruction (putstatic t n) st cap) in
  ((x:xLabel) ~res=(ThrowException x st cap)) →
  (Same_static_heap st cap) →
  (In (alpha_da_rdstate res cap)
                  (aexec_instruction (putstatic t n)
                            (alpha_da_state st cap)
                             cap)).
```

On remarque ainsi que l'on construit explicitement ligne 3 un état avec le contexte d'exécution `h` en tête de pile d'exécution. Dans le cas de `putstatic`, nous n'avons qu'à rejeter les exécutions défensives menant à l'appel de `ThrowException`. La ligne 6 introduit une hypothèse garantissant que les deux machines virtuelles considèrent le même tas statique.

4.9 De la machine virtuelle abstraite au vérifieur de bytecode

Les travaux de G. Dufay et de G. Barthe [17, 22, 59] considèrent la construction d'un vérifieur de bytecode basé sur la machine virtuelle abstraite. La construction reprend les principes de l'algorithme de Kildall et celle-ci fait l'objet d'une preuve de correction : d'une part l'algorithme est démontré terminant, d'autre part il est démontré que si la machine défensive lance une erreur de type, alors le BCV lui aussi la signale. Le module BCV considère aussi la preuve de correction de la machine virtuelle abstraite pour les instructions modifiant la pile d´exécution.

4.10 Conclusion

Nous avons présenté dans ce chapitre une formalisation Coq de plusieurs machines virtuelles JavaCard : une machine défensive, une machine offensive et une machine abstraite. Les outils développés autour de la formalisation permettent de plus de calculer la version Coq de tout programme JavaCard.

Complétude relative Notre machine virtuelle est relativement complète dans le sens où elle comprend la modélisation de tout l'ensemble d'instructions JavaCard et prend en compte les aspects importants de la plate-forme, incluant le pare-feu et les mécanismes de partage d'objets autour desquels la sécurité JavaCard est organisée. Cependant nous ne considérons pas tous les aspects du JCRE, tels que les APIs propres aux transactions et aux objets transients.

Évaluation de CertiCartes Nos conclusions sur l'effort de formalisation de CertiCartes rejoignent celles de l'équipe de Trusted Logic à propos de leur travaux de formalisation [74] : Si les assistants de preuve sont maintenant arrivés à un niveau de maturité qui en permet l'utilisation dans un cadre industriel, ceux-ci manquent cependant de fonctionnalités permettant de supporter au mieux les développements de grande taille.

Précisons ce point, considérons la phase de définition d'une spécification. D'une part le langage formel propre à ces systèmes impose une rigueur, au demeurant saine, mais alourdissant la clarté des grandes spécifications. D'autre part l'écriture de variations de la spécification originale — les machines aJCVM et oJCVM à partir de la dJCVM, s'est révélée systématique mais cependant fastidieuse. Coq, et en général les assistants de preuve, ne fournit pas un support particulier à ce type de situation, pourtant courant dans le cadre de la vérification formelle de plates-formes logicielles.

Du point de vue des preuves, la situation est semblable. La preuve de commutation de deux exécutions est en général très répétitive. Seuls certains pas, de fait les pas pertinents, nécessitent une attention particulière. On pourrait argumenter que c'est le prix à payer pour le pouvoir d'expression des assistants de preuve. Cependant la nature systématique de la preuve des cas non pertinents suggère la possibilité de l'automatisation. C'est ce point de vue qui a conduit les travaux de P. Courtieu [16].

Analysons maintenant la formalisation elle-même. CertiCartes est une formalisation complète de la plate-forme JavaCard qui constitue une étude de faisabilité approfondie d'évaluation EAL 7. Les laboratoires de recherche de GEMPLUS ont effectué à ce propos une évaluation de CertiCartes [101]. Leur constat général est que la méthodologie employée est adéquate. Sur CertiCartes en particulier, l'évaluation rapporte que, si la portée de CertiCartes est suffisante, le grain n'en est pas suffisamment fin. Ceci est essentiellement dû aux points suivants :

- le traitement monolithique de la mémoire qui ne permet pas ainsi de considérer les objets transients.
- l'absence du traitement des transactions ;
- l'absence du traitement des méthodes natives, bien que G. Dufay en ait indiqué la méthode d'intégration (voir [17]).

La conclusion est que l'intégration de ces éléments ne semble pas problématique mais exige un effort (d'implémentation et non pas conceptuel) non négligeable.

Travaux futurs Outre le raffinement de la machine défensive que nous avons introduit à la section précédente et qui nous permettrait de mieux nous rapprocher du cadre d'une certification EAL7, soulignons une autre direction de recherche que nous aimerions développer autour de CertiCartes : la preuve de programme.

En effet, la représentation Coq des programmes JavaCard peut servir de base au raisonnement sur les programmes eux-mêmes. En particulier, il est possible d'extraire de notre formalisation les traces d'exécution d'un programme, et dès lors de raisonner sur les propriétés de sécurité qui sont souvent exprimées en terme de propriétés de logique temporelle sur les traces d'exécution [138].

JaKarTa, que nous présentons dans le chapitre suivant, est un outil permettant de palier les problèmes de manque de systématisation que nous avons soulignés, tant au niveau des spécifications qu'au niveau des preuves. Cet outil permet d'envisager la construction de nouvelles abstractions de machines virtuelles. Sont visées les abstractions permettant de mieux appréhender les problèmes de non-interférence et les problèmes d'initialisation des objets.

Chapitre 5

JaKarTa : un outil de support à la spécification et à la vérification formelle

Ce chapitre fournit une description détaillée de l'outil JaKarTa [19, 20]. Cet outil apporte une solution possible aux difficultés soulevées par l'approche CertiCartes en assemblant systèmes de preuve, environnement de prototypage et calcul de spécification.

Objectifs Les objectifs poursuivis par JaKarTa sont essentiellement de fournir un support automatique à la construction de machines virtuelles, pour une propriété arbitraire P, et de mettre en place l'encadrement nécessaire à la validation croisée de ces machines virtuelles.

Cahier des charges Afin de faciliter le support à la transformation de spécifications, l'outil JaKarTa a été implémenté en suivant le cahier des charges suivant :

– *Flexibilité des spécifications* : les spécifications JaKarTa sont écrites dans le langage JSL (pour *JaKarTa Specification Language*). Ce format permet la définition de spécifications exécutables. C'est un langage de premier ordre avec un système de types polymorphes, dans lequel les fonctions sont définies par des règles de réécriture conditionnelle. Les spécifications produites sont raisonnablement faciles à lire et, le plus important, à manipuler. Le module chargé du calcul de transformations de spécifications est appelé JTK (pour *JaKarTa Transformation Kit*).
  Concrètement nous voulons que le JTK puisse, à partir des données suivantes :
  - une machine virtuelle P-défensive `dexec: dstate` $\rightarrow$ `dstate` qui manipule des valeurs typées (la notion de type étant déterminée par P) et qui garantit P par une vérification dynamique (au cours de l'exécution) ;
  - deux fonctions d'abstraction :

    ```
    alpha_da : dstate → astate
    alpha_do : dstate → ostate
    ```

    où `astate` et `ostate` dénotent les types des états abstraits et offensifs adéquats ;

- un script de transformation, c'est-à-dire une séquence de commandes indiquant comment conduire la dérivation de la machine P-défensive vers les machines P-offensive et P-abstraite ;

produire une machine P-offensive et une machine P-abstraite :

```
oexec : ostate → rostate
aexec : astate → (list rastate)
```

Un autre avantage de la flexibilité du langage JSL est la possibilité d'intégrer la définition de fonctions partielles, comme la tête de liste, et de fonctions non-déterministes au sein du langage. Bien que les fonctions ne puissent être traduites telles quelles vers les langages des systèmes de preuve, il est possible de les transformer automatiquement, via le JTK, en des fonctions dont la traduction est acceptée par ces systèmes (voir section 5.2.4).

– *Neutralité* : Le langage de spécification JSL est minimal. Il est dès lors facile de traduire les spécifications JSL vers d'autres formalismes. Le module JPI (pour *JaKarTa Prover Interface*) permet de telles traductions directement à partir du format JSL, ou via une extension de ce format, appelée JIR(pour *JaKarTa Intermediate Representation*) . Les formalismes cibles sont les langages de programmation ou encore les langages propres aux systèmes de preuves et environnement de prototypage. Parce que le format des règles de réécriture JSL est restreint, les spécifications JSL sont suffisamment proches de leur traduction dans les systèmes de preuve. Ceci permet d'assurer, par exemple, que l'on raisonne bien dans Coq sur ce qui a été spécifié en JSL.
– *Automatisation des preuves* : toujours grâce au format restreint de JSL, il est possible de développer des outils permettant le raisonnement sur les spécifications JSL. Le module JAK (pour *JaKarTa Automation Kit* implémente ces outils et les propose au système Coq. Plus précisément JAK génère pour chaque fonction JSL f un principe d'inversion qui est basé sur une analyse des règles de réécriture définissant f.
  Soit ϕ le prédicat reliant les entrées de f aux sorties de f. L'application du principe d'inversion réduit la preuve de l'énoncé $\forall \vec{x} : \vec{\sigma}.\ \phi\ (\vec{x},\ f\ \vec{x})$ en la preuve de plusieurs énoncés plus simples.
  A l'encontre de la généralité du JPI, soulignons que les outils du module JAK produisent en sortie des données qui ne sont exploitables que par Coq.
  Le module JaKarTa a aussi été pensé pour contribuer à la validation croisée des machines synthétisées. Pour chaque fonction de la spécification originale qui a subi une transformation, un lemme de commutation et une preuve utilisant le principe d'inversion cité sont produits. Ce module permet ainsi d'associer à chaque abstraction calculée la preuve qu'elle résulte de transformations cohérentes de la machine originale.

L'outil JaKarTa est implémenté en Objective Caml [108]. La figure 5.1 résume l'architecture et l'utilisation de JaKarTa. Pour commencer, JaKarTa prend en entrée une spécification JSL. Le Kit de Transformation JaKarTa (JTK) est utilisé pour obtenir une spécification abstraite calculée à partir de la spécification originale. L'Interface de Preuve JaKarTa (JPI) traduit les spécifications JSL vers des formats externes à l'outil, éventuellement via la Représentation Intermédiaire JaKarTa (JIR) des spécifications JSL. Tandis que le Kit d'Automatisation JaKarTa (JAK) génère en Coq des stratégies de preuve adéquates pour le raisonnement sur les spécifications JSL.

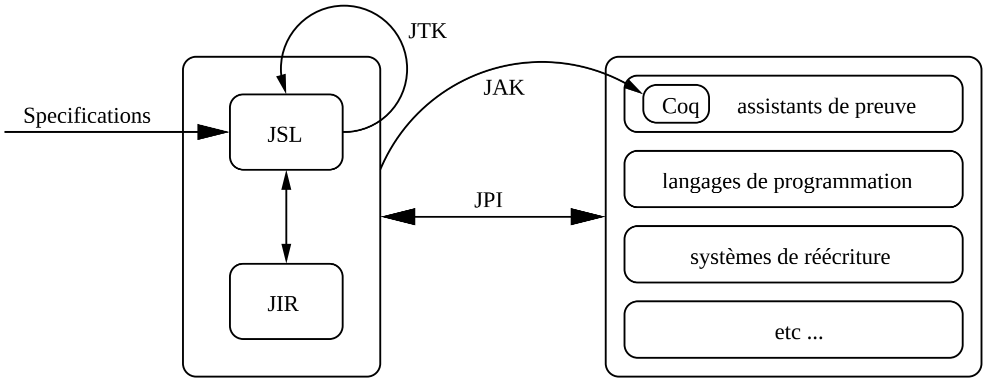

FIG. 5.1 – Architecture de l'outil JaKarTa.

Organisation du chapitre Le reste du chapitre est organisé de la façon suivante : la section 5.1 introduit le langage JSL ainsi que son système de types. La section 5.2 décrit les fondements et la mise en place des mécanismes d'abstractions fournis par le JTK. La section 5.3 expose les fonctionnalités d'interaction de JaKarTa avec les systèmes externes telles qu'elles sont implémentées par le module JPI. Finalement, la section 5.4 présente le support à la preuve donné par JAK.

5.1 JSL : support à la spécification

Le support à la spécification offert par JaKarTa se compose d'un langage de spécification, appelé **JSL** (pour JaKarTa Specification Language), et d'un vérificateur de contraintes liées au langage proposé, comme par exemple les contraintes de types.

Ce langage permet la définition de types de données inductifs polymorphes de premier ordre, la définition de fonctions par règles de réécriture où les expressions sont des termes du premier ordre avec égalité (notée ==).

Dans un premier temps nous décrirons la formation syntaxique des éléments du langage. Nous commencerons par les termes du langage et les termes de type. Ensuite seront introduites les règles de réécriture conditionnelle. Nous serons alors en mesure de définir ce qu'est une spécification JSL. Finalement nous décrirons les contraintes liées au typage des définitions de fonction.

Par souci de clarté, nous n'aborderons pas dans ce chapitre l'existence des définitions mutuelles qui n'ont pas été, au demeurant, difficiles à mettre en place, mais qui, dans le cadre de cette exposition, nuisent à la simplicité.

5.1.1 La syntaxe abstraite JSL

Nous introduisons dans cette section la syntaxe abstraite du langage JSL. Dans ce cadre, les détails, les contraintes, propres à la syntaxe concrète et à une implémentation du langage ne sont pas pertinents. Néanmoins, nous essayerons de transposer au sein même de la syntaxe abstraite certaines de ces contraintes — les *contraintes de bonne formation*, car elles jouerons un rôle dans le calcul d'abstraction.

Les éléments du langage JSL reposent sur deux ensembles disjoints énumérables : un ensemble de variables de termes $\mathcal{V}$ et un ensemble de variables de type $\mathcal{V}_{\mathcal{T}}$.

On distingue deux classes de termes dans JSL : les expressions et les types. Les expressions sont des termes du premier ordre construits à partir de variables de terme et de symboles de constante. Ces derniers sont soient des symboles de constructeur, introduits par des définitions de types de données, soient des symboles de fonction, introduits par des définitions de fonctions.

Définition 1 (Expressions)
- *Soient $\mathcal{C}$ un ensemble de symboles de constructeur, $\mathcal{D}$ un ensemble de symboles de fonction,* $\mathrm{ar} : (\mathcal{C} \cup \mathcal{D}) \to \mathbb{N}$ *une fonction arité.*
- *L'ensemble des expressions est l'ensemble $\mathcal{E}$ des termes défini par la syntaxe suivante :*
$$\mathcal{E} ::= \mathcal{V} \mid \mathcal{E} == \mathcal{E} \mid \mathcal{C}\ \mathcal{E}^* \mid \mathcal{D}\ \mathcal{E}^*$$
où, dans $(u\ t_1 \ldots t_n)$ avec $u \in \mathcal{C} \cup \mathcal{D}$, il est requis que $n = \mathrm{ar}(u)$.

Exemples Supposons que nous ayons les ensembles $\mathcal{V} = \{x, l\}$, $\mathcal{C} = \{True,\ False,\ Zero,\ Succ,\ Nil,\ Cons\}$ et $\mathcal{D} = \{plus,\ head,\ tail\}$ et la fonction $ar\ : \{(True \mapsto 0), (False \mapsto 0), (Zero \mapsto 0), (Succ \mapsto 1), (Nil \mapsto 0), (Cons \mapsto 2), (plus \mapsto 2), (head \mapsto 1), (tail \mapsto 1)\}$.
- les expressions suivantes appartiennent à $\mathcal{E}$: $(plus\ x\ (Succ\ Zero))$, $(Cons\ (head\ l)\ (tail\ l))\ Zero\ ==\ (Cons\ Zero\ Nil)$. En effet tous les symboles appartiennent à $\mathcal{C} \cup \mathcal{D}$ et les contraintes d'arité sont respectées.
- les expressions suivantes n'appartiennent pas à $\mathcal{E}$: $(pred\ x)$ car $pred$ n'est pas un symbole de $\mathcal{C} \cup \mathcal{D}$. $(plus\ Zero)$, car $plus$ est binaire.

Les expressions de type sont construites à partir :
- d'un ensemble de variables de type $(\mathcal{V}_{\mathcal{T}})$;
- d'un ensemble de symboles de type. Ceux-ci sont des symboles soient de type de données inductif $(\mathcal{T}_d)$, soient de synonyme de type $(\mathcal{T}_s)$ ou soient de type abstrait$(\mathcal{T}_a)$;
- du symbole $\to$, qui permet la formation des types de fonctions ou de constructeurs et
- du symbole $\forall$ qui permet d'introduire le polymorphisme.

Définition 2 (types)

- *Soient* $\mathcal{T} = \mathcal{T}_d \cup \mathcal{T}_s \cup \mathcal{T}_a$ *l'ensemble des symboles de type,* $ar_{\mathcal{T}} : \mathcal{T} \to \mathbb{N}$ *une fonction arité sur les éléments de* $\mathcal{T}$*.*
- *L'ensemble des types élémentaires* $\mathcal{T}y$ *est l'ensemble des termes définis par la syntaxe suivante :*
$$\mathcal{T}y ::= \ \mathcal{V}_{\mathcal{T}} \mid \ \mathcal{T}\ \mathcal{T}y^*$$
où, dans $(t\ t_1 \ldots t_n)$ *avec* $t \in \mathcal{T}y$*, il est requis que* $n = \mathrm{ar}_{\mathcal{T}}(t)$ *;*
- *L'ensemble* $\mathcal{E}_{\mathcal{T}\to}$ *des types élémentaires clos par* $\to$ *est défini par :*
$$\mathcal{E}_{\mathcal{T}\to} ::= \mathcal{T}y \ \mid \ \mathcal{T}y \to \mathcal{E}_{\mathcal{T}\to}$$
- *L'ensemble des expressions de types* $\mathcal{E}_{\mathcal{T}}$ *est l'ensemble des termes de* $\mathcal{E}_{\mathcal{T}\to}$ *clos par* $\forall$ *:*
$$\mathcal{E}_{\mathcal{T}} ::= \ \forall \mathcal{V}_{\mathcal{T}}.\ \mathcal{E}_{\mathcal{T}} \ \mid \ \mathcal{E}_{\mathcal{T}\to}$$
- *L'ensemble* $\mathcal{S}_{\mathcal{T}} \subset \mathcal{E}_{\mathcal{T}}$ *des types de fonctions et de constructeurs est défini comme l'ensemble des expressions de type closes — c'est-à-dire ne contenant pas de variable libre.*

Nous supposerons dans la suite — c'est une contrainte de bonne formation, que les ensembles $\mathcal{V}$, $\mathcal{V}_{\mathcal{T}}$, $\mathcal{C}$, $\mathcal{D}$, $\mathcal{T}_d$, $\mathcal{T}_s$ et $\mathcal{T}_a$ sont mutuellement disjoint.

Nous supposerons dans ce qui suit que le lecteur est accoutumé aux notions standard de sous-terme, d'occurrences, substitutions ainsi qu'aux notations associées. Par exemple, on notera par $\mathsf{var}(t)$ (resp. $\mathsf{var}_{\mathcal{T}}(ty)$) l'ensemble des variables d'une expression t (resp. d'un type ty). Consulter *e.g.* [98] dans le cas contraire.

Exemples Nous pouvons considérer les ensembles suivants :

$$\begin{array}{lcl}
\mathcal{T}_d & = & \{pair,\ list,\ nat,\ bool\} \\
\mathcal{T}_s & = & \{list_nat,\ sequence\} \\
\mathcal{T}_a & = & \{rationnel,\ arbre\} \\
ar_{\mathcal{T}} & = & \{(bool \quad \mapsto \ 0), \\
 & & \ \ (pair \quad \mapsto \ 2), \\
 & & \ \ (list \quad \mapsto \ 1), \\
 & & \ \ (nat \quad \mapsto \ 0), \\
 & & \ \ (list_nat \quad \mapsto \ 0), \\
 & & \ \ (sequence \quad \mapsto \ 1), \\
 & & \ \ (rationnel \quad \mapsto \ 0), \\
 & & \ \ (arbre \quad \mapsto \ 1)\}
\end{array}$$

Dans cette configuration, les expressions de types suivantes sont correctes :
- $(list\ (pair\ nat\ nat))$ est le type des listes de paires d'entiers ;
- $\forall \alpha. \forall \beta. nat \to (list\ \alpha) \to (list\ \beta) \to (list\ (pair\ \alpha\ \beta))$ appartient à $\mathcal{S}_{\mathcal{T}}$ et est le type d'une fonction qui prend un entier, une liste d'éléments d'un type arbitraire α, une liste d'éléments d'un type arbitraire β et rend une liste de type $(list\ (pair\ \alpha\ \beta))$. C'est par exemple le type de la fonction `zip_n` qui assemble les éléments de deux listes en une liste de paires de longueur prédéfinie (par le premier argument).

Règles JSL

Les symboles de fonction sont définis par des règles de réécriture conditionnelle. L'idée sous-jacente à l'utilisation de la réécriture conditionnelle est de fournir un format de définition de fonctions qui soit à la fois souple et neutre, mais aussi suffisamment proche des langages fonctionnels avec filtrage. Ceci, essentiellement, pour les raisons suivantes :

1. permettre une traduction aisée de ce format vers les langages à filtrage de motif ;
2. disposer d'une représentation où la notion de chemin est plus explicite que dans la notion de représentation arborescente des différents filtrages. Ce point est particulièrement important dans le cadre de la transformation de spécification où l'on considère souvent une fonction selon les valeurs que ses variables peuvent prendre. Un chemin, ou branche, représente exactement une configuration possible de valeurs des variables de la fonction considérée. Sous cet angle, parler de branche est beaucoup plus convenable que de parler de filtrages de motifs imbriqués ;
3. la représentation par règles de réécriture conditionnelle est plus permissive que la représentation arborescente. Elle nous permet, par exemple, l'expression de fonctions non-déterministes.

Cette similitude avec les langages fonctionnels est assurée par un certain nombre de contraintes syntaxiques. Plus précisément, la sémantique d'un symbole de fonction défini est décrite par un ensemble de règles de réécriture conditionnelle orientée de la forme :

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

où il est requis que les membres droits des conditions $l_i \twoheadrightarrow r_i$ soient des expressions particulières, appelées motifs. En particulier, dans une condition $l \twoheadrightarrow r$, l doit représenter un appel de fonction et ne doit pas introduire de variables, et

- r doit représenter une valeur (ce que nous appellerons motif),
- r ne doit contenir que des variables fraîches
- r doit être linéaire.

Ceci permet de rapprocher une règle $l_1 \twoheadrightarrow r_1,\ \ldots\ , l_n \twoheadrightarrow r_n \Rightarrow g \rightarrow d$ d'un chemin, de la racine aux feuilles, d'une définition par filtrage qui est arborescente. Par exemple les règles

$$l \twoheadrightarrow Nil \quad \Rightarrow \quad (is\_empty\ l) \rightarrow True \tag{5.1}$$

$$l \twoheadrightarrow (Cons\ \ el\ tl) \quad \Rightarrow \quad (is\_empty\ l) \rightarrow False \tag{5.2}$$

correspondent respectivement à la première branche et à la deuxième branche de la fonction Caml suivante :

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

Définition 3 (Motifs)
L'ensemble $\mathcal{P}$ des motifs est le sous-ensemble de $\mathcal{E}$ défini par la syntaxe abstraite suivante :

$$\mathcal{P} := \mathcal{V} \mid \mathcal{C}\ \mathcal{P}^*$$

*où il est requis que, dans la seconde clause, les motifs aient un ensemble de variables disjointes (**i.e.** les motifs vérifient la propriété de linéarité) ;*

Définition 4 (Règles de réécriture)

- *Pour chaque $f \in \mathcal{D}$, soit $\mathcal{F}(f) \subset \mathcal{E}$ l'ensemble des expressions ayant f comme symbole de tête,* i.e. $\mathcal{F}(f) = \{f\ \vec{t} \mid \vec{t} \in \vec{\mathcal{E}}\}$. *Définissons alors $\mathcal{F}$ par $\mathcal{V} \cup (\bigcup_{f\in\mathcal{D}} \mathcal{F}(f))$;*
- *L'ensemble $\mathcal{R}$ des règles de réécriture est défini par*

$$\mathcal{R} = (\mathcal{F} \times \mathcal{P})^* \times (\mathcal{F} \times \mathcal{E})$$

- *Étant donnés un symbole de fonction f, une f*-règle de réécriture *R, $R \in \mathcal{R}$, est de la forme :*

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

  où :
  - *$g = f\ \vec{x}$ où $\vec{x} \in \mathcal{V}$ sont deux à deux disjoints ;*
  - *pour $1 \leq i \leq n$, $\mathsf{var}(l_i) \subseteq \mathsf{var}(g) \cup \mathsf{var}(r_1) \cup\ \ldots\ \cup \mathsf{var}(r_{i-1})$;*
  - *pour $1 \leq i, j \leq n$ avec $i \neq j$, $\mathsf{var}(r_i) \cap \mathsf{var}(g) = \emptyset$ et $\mathsf{var}(r_i) \cap \mathsf{var}(r_j) = \emptyset$;*
  - *$\mathsf{var}(d) \subseteq \mathsf{var}(g) \cup \mathsf{var}(r_1) \cup\ \ldots\ \cup \mathsf{var}(r_n)$.*

  Nous noterons l'ensemble des f-règles de réécriture *par $\mathcal{R}(f)$;*
- *Le* domaine $\mathsf{dom}(R)$ *d'une règle de réécriture R est défini par*

$$\mathsf{var}(g) \cup \bigcup_{1\leq i\leq n} \mathsf{var}(r_i)$$

- *L'ensemble des systèmes de réécriture noté $\mathbb{R}$, est défini par*[1] $\mathbb{R} = \mathcal{P}_f(\mathcal{R})$.

En prenant comme référence [27], la notion de réécriture conditionnelle pour les règles est formellement définie de la façon suivante :

Définition 5 (Modèle d'exécution JSL)
Soit R un ensemble de règles de réécriture. Une expression s se R-réécrit en t, noté $s \rightarrow_R t$, s'il existe une règle $r \in R$

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

une position p dans s et une substitution θ ayant comme domaine $\mathsf{dom}(r)$ *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 $\rightarrow_R$.

[1] $\mathcal{P}_f(E)$ est la notation pour l'ensemble des parties finies d'un ensemble E

Relations de précédence et contraintes de bonne formation

Nous admettrons l'existence des trois relations de précédence suivantes sur les ensembles $\mathcal{T}$, $\mathcal{C} \cup \mathcal{D}$ et $\mathcal{R}$:
- L'ordre $\leq_{\mathcal{T}}$ sur les symboles de types ;
- L'ordre $\leq_{\mathcal{F}}$ sur les symboles de fonction et de constructeur ;
- L'ordre $\leq_{\mathcal{R}}$ sur les règles de réécriture.

Soulignons que les deux derniers ordres seront utiles à la définition de l'ordre sur les positions des expressions (Définition 9).

Intuitivement et dans le cadre d'une implémentation de JSL, l'ordre de définition des différents éléments du langage induit ces relations de précédence. Toute définition ne peut se référer qu'aux symboles déjà définis. Ces contraintes de précédence sont un sous-ensemble des contraintes de bonne formation, au même titre que les contraintes d'arité des expressions, que nous avons référées en début de section.

Ainsi, à la définition des différents éléments du langage JSL, nous ajoutons les contraintes additionnelles suivantes :
- toute définition de fonction ou de constructeur respecte $\leq_{\mathcal{F}}$;
- toute définition de type t respecte la précédence induite par $\leq_{\mathcal{T}}$.

Quelques notations et remarques

Notons qu'aucune contrainte particulière n'aborde l'exhaustivité des cas traités par un ensemble de f-règles de réécriture, de même aucune contrainte n'oblige que ce système de règles soit confluent. La conséquence immédiate est la possibilité de définir des fonctions partielles et non déterministes.

Dans la suite de ce texte, nous utiliserons la notion de graphe de dépendance d'une fonction JSL. Par ce terme, nous désignons, de façon tout à fait usuelle, la fermeture transitive de la relation “fait appel à”.

Soit $c = l \twoheadrightarrow r$, une condition. Nous serons amenés à utiliser les notations $lhs(c)$ et $rhs(c)$ pour désigner le membre gauche l et le membre droit r de la condition c. Soit R la règle $l_1 \twoheadrightarrow r_1,\ \ldots\ ,\ l_n \twoheadrightarrow r_n \Rightarrow g \rightarrow d$, la notation $R(i)$ avec $1 \leq i \leq n$ désignera la condition $l_i \twoheadrightarrow r_i$.

Soit R la règle $l_1 \twoheadrightarrow r_1,\ \ldots\ ,\ l_n \twoheadrightarrow r_n \Rightarrow g \rightarrow d$. Nous désignerons par corps de R (noté $corps(R)$) la séquence de conditions $l_1 \twoheadrightarrow r_1,\ \ldots\ ,\ l_n \twoheadrightarrow r_n$ et par conclusion (notée $conclusion(R)$) l'expression d.

5.1.2 Spécification JSL

Définition 6 (Signature d'expressions) *Une signature d'expressions Σ consiste en un ensemble $\mathcal{C}$ de symboles de constructeur, en un ensemble $\mathcal{D}$ de symboles de fonction, en une fonction arité $ar\ :\ (\mathcal{C} \cup \mathcal{D}) \rightarrow \mathbb{N}$, en une relation de précédence $\leq_{\mathcal{F}}$ sur les symboles de fonction, en une relation de précédence $\leq_{\mathcal{R}}$ sur les règles et en la fonction* rules $: \mathcal{D} \rightarrow \mathbb{R}$ *qui à tout symbole de fonction f lui fait correspondre l'ensemble $\mathcal{R}(f)$ de ses $f-$règles :*

Nous avons donc :

$$\Sigma = \{(\mathcal{C} \cup \mathcal{D}), \leq_{\mathcal{F}}, \leq_{\mathcal{R}}, ar, rules\}$$

Soit f un symbole de fonction, dans le cas où $rules(f) = \emptyset$, nous dirons que f est une fonction variable.

Définition 7 (Signature de types) *Une signature de types $\Sigma_{\mathcal{T}}$ consiste en l'ensemble $\mathcal{T}$, en la fonction ar, en la relation $\leq_{\mathcal{T}}$ et les fonctions suivantes :*

- *constructors : $\mathcal{T}_d \rightarrow \mathcal{P}_f(\mathcal{C})$ qui à tout symbole de type de données fait correspondre l'ensemble des symboles de ses constructeurs ;*
- *synonym : $\mathcal{T}_s \rightarrow \mathcal{T}_\mathcal{Y}$ qui à tout symbole de $\mathcal{T}_s$ de type fait correspondre l'expression de type de laquelle il est synonyme ;*
- *type : $\mathcal{C} \cup \mathcal{D} \rightarrow \mathcal{S}_\mathcal{T}$ qui à tout symbole de fonction ou de constructeur lui fait correspondre son type.*

$$\Sigma_{\mathcal{T}} = \{\mathcal{T} = (\mathcal{T}_d \cup \mathcal{T}_s \cup \mathcal{T}_a),\ \leq_{\mathcal{T}},\ \ ar_{\mathcal{T}},\ constructors, synonym,\ type\}$$

Nous sommes enfin en mesure de préciser ce que nous entendons par spécification JSL : c'est une collection de types, de constructeurs et de fonctions.

Définition 8 (Spécification JSL) *Une spécification JSL, S_{pec} est la donnée d'une signature d'expressions Σ et d'une spécification de types $\Sigma_{\mathcal{T}}$:*

$$S_{pec} = \{\Sigma, \Sigma_{\mathcal{T}}\}$$

Position d'un terme dans une spécification JSL

Insistons sur quelques particularités propres à la notion de position $pos(t)$ d'un terme t dans une spécification JSL. Elle sera intensivement utilisée par le JTK lorsque celui-ci cherchera à manipuler des termes en fonction de leur position (voir sections 5.2.2 et 5.2.6). Celle-ci est l'extension naturelle de la notion de position dans un terme à la spécification toute entière. Ainsi on se permet de parler du premier sous-terme du membre gauche de la première condition de la première règle définissant la fonction f. La forme compacte d'une telle position est f.1.1.1.1.

La particularité de la notion de position adoptée vient de l'ordre sur les positions. L'ordre est standard pour toute comparaison de positions d'expression sauf pour celles impliquant des membres de conditions. Explicitons ce point.

Définition 9 (Ordre sur les positions)

Considérons les ordres suivants :

- *l'ordre $\leq_{\mathcal{F}}$ sur les symboles de fonction ;*
- *l'ordre $\leq_{\mathcal{R}}$ sur les règles ;*
- *l'ordre $\leq_m$ sur les membres de condition défini de la façon suivante : Soient f une fonction et R la $f-$règle*

$$m_1 \twoheadrightarrow m_2,\ \ldots\ ,\ m_n \twoheadrightarrow m_{n+1} \Rightarrow m_0 \rightarrow m_{n+2}$$

Soit $(i, j) \in \mathbb{N}^2$ tels que $0 \leq i \leq j \leq n+2$, alors $m_i \leq_m m_j$.

En d'autres mots, la séquence utilisée pour comparer la position de membres dans R *est* m_o, m_1, $\ldots$ m_n, m_{n+1}, m_{n+2} *et non pas* m_1, $\ldots$ m_n, m_{n+1}, m_0, m_{n+2} *;*

– *l'ordre standard* $\leq_{\mathcal{T}}$ *sur les (sous-)termes.*

Alors, l'ordre $\leq$ *sur les positions est défini comme l'ordre lexicographique* $(\leq_{\mathcal{F}}, \leq_{\mathcal{R}}, \leq_m, \leq_{\mathcal{T}})$.

Exemples

Nous présentons 5.2 et 5.3 un exemple complet et représentatif de spécification JSL.

Nous introduisons, dans cet exemple, le type des paires polymorphes *pair*, le type des listes polymorphes *list*, le type des entiers naturels *nat* et le type des booléens *bool* comme des types inductifs. Nous accompagnons la définition de ces types par la définition de leurs constructeurs : $True$, $False$, $Zero$, *Succ*, Nil, *Cons*, $MkPair$, ainsi que de quelques fonctions usuelles : *plus*, $left$, *right*.

Nous définissons aussi un type synonyme, *list_nat*, des listes d'entiers et un synonyme des listes, *sequence*, qui reste polymorphe sur le type des éléments.

Pour illustrer la définition de types abstraits, nous introduisons les types *rationnel* et *arbre*, de façon à ce que ce dernier soit polymorphe. Nous déclarons aussi une fonction *is_empty* sur le type abstrait des arbres.

Remarquons que la fonction *plus* est récursive et que la fonction *is_empty* est déclarée comme une fonction variable. Ceci s'explique par le fait qu'elle est définie sur le type abstrait *arbre*.

Nous avons défini dans cette section chacun des éléments du langage JSL. Ces définitions introduisent des contraintes syntaxiques sur la forme que peuvent prendre ces définitions. Nous avons vu que ces contraintes dépendent fortement des ensembles Σ et $\Sigma_{\mathcal{T}}$, des relations de précédence et du format désiré pour chacun des éléments du langage. Nous nous référerons à cet ensemble de contraintes par *contraintes de bonne formation*.

Nous allons introduire dans la section suivante un autre type de contraintes : les *contraintes de typage*.

5.1.3 Vérification du typage

La vérification du typage est essentiellement la vérification, pour une expression donnée, de la concordance du type inféré avec le type attendu. Cette vérification de concordance s'exprime algorithmiquement en terme d'inférence de type et d'unification d'expressions de type. Cependant afin de garder une présentation claire d'une notion somme toute simple, nous ne nous intéresserons ici qu'à la définition formelle de la notion de typage sans aborder les aspects propres à l'implantation.

Environnement et contexte Soit $S_{pec} = \{\Sigma, \Sigma_{\mathcal{T}}\}$ une spécification JSL. Une définition classique de la propriété de bon typage d'une expression a besoin de considérer l'existence de deux ensembles. Le premier, l'environnement que

Soit $\Sigma = \{(\mathcal{C} \cup \mathcal{D}), \leq_{\mathcal{F}}, \leq_{\mathcal{R}}, ar, rules\}$ la signature d'expressions suivante :

$$
\begin{array}{lcl}
\mathcal{D} & = & \{plus,\ left,\ right,\ is\_empty\} \\
\mathcal{C} & = & \{True,\ False,\ Zero,\ Succ,\ Nil,\ Cons,\ MkPair\} \\
\leq_{\mathcal{F}} & = & plus \leq_{\mathcal{F}} left \leq_{\mathcal{F}} right \leq_{\mathcal{F}} is\_empty \\
\\
ar & = & \begin{array}[t]{lcl}
\{(True & \mapsto & 0), \\
\ (False & \mapsto & 0), \\
\ (Zero & \mapsto & 0), \\
\ (Succ & \mapsto & 1), \\
\ (Nil & \mapsto & 0), \\
\ (Cons & \mapsto & 2), \\
\ (MkPair & \mapsto & 2), \\
\ (left & \mapsto & 1), \\
\ (right & \mapsto & 1), \\
\ (plus & \mapsto & 2), \\
\ (is\_empty & \mapsto & 1)\}
\end{array} \\
\\
rules & = & \begin{array}[t]{lcl}
\{(plus & \mapsto & \{(x \twoheadrightarrow Zero \Rightarrow plus\ x\ y \rightarrow y), \\
 & & \ (x \twoheadrightarrow Succ\ m \Rightarrow \\
 & & \qquad plus\ x\ y \rightarrow Succ\ (plus\ m\ y))\}), \\
\ (is\_empty & \mapsto & \emptyset), \\
\ (left & \mapsto & \{x \twoheadrightarrow (MkPair\ a\ b) \Rightarrow left\ x \rightarrow a\}), \\
\ (right & \mapsto & \{x \twoheadrightarrow (MkPair\ a\ b) \Rightarrow right\ x \rightarrow b\})\}
\end{array} \\
\\
\leq_{\mathcal{R}} & = & (x \twoheadrightarrow Zero \Rightarrow plus\ x\ y \rightarrow y) \leq_{\mathcal{R}} \\
 & & (x \twoheadrightarrow Succ\ m \Rightarrow plus\ x\ y \rightarrow Succ\ (plus\ m\ y))
\end{array}
$$

FIG. 5.2 – Exemple d'une signature d'expressions

Soit $\Sigma_{\mathcal{T}} = \{\mathcal{T} = (\mathcal{T}_d \cup \mathcal{T}_s \cup \mathcal{T}_a),\ \leq_{\mathcal{T}},\ \ ar_{\mathcal{T}},\ constructors,$ $synonym,\ type\}$ la signature de types suivante :

$$\begin{array}{lcl}
\mathcal{T}_d & = & \{pair,\ list,\ nat,\ bool\} \\
\mathcal{T}_s & = & \{list_nat,\ sequence\} \\
\mathcal{T}_a & = & \{rationnel,\ arbre\} \\
\leq_{\mathcal{T}} & = & \{bool \leq_{\mathcal{T}} nat \leq_{\mathcal{T}} list \leq_{\mathcal{T}} pair \leq_{\mathcal{T}} list_nat \\
& & \quad \leq_{\mathcal{T}} sequence \leq_{\mathcal{T}} rationnel \leq_{\mathcal{T}} arbre\}
\end{array}$$

$$\begin{array}{lclcl}
ar_{\mathcal{T}} & = & \{(bool & \mapsto & 0), \\
 & & \ (pair & \mapsto & 2), \\
 & & \ (list & \mapsto & 1), \\
 & & \ (nat & \mapsto & 0), \\
 & & \ (list_nat & \mapsto & 0), \\
 & & \ (sequence & \mapsto & 1), \\
 & & \ (rationnel & \mapsto & 0), \\
 & & \ (arbre & \mapsto & 1)\} \\
 & & & & \\
constructors & = & \{(bool & \mapsto & \{True,\ False\}), \\
 & & \ (nat & \mapsto & \{Zero,\ Succ\}), \\
 & & \ (pair & \mapsto & \{MkPair\}), \\
 & & \ (list & \mapsto & \{Nil,\ Cons\})\} \\
 & & & & \\
synonym & = & \{(list_nat & \mapsto & (list\ nat)), \\
 & & \ (sequence & \mapsto & \forall\alpha.(list\ \alpha))\} \\
 & & & & \\
type & = & \{(True & \mapsto & bool), \\
 & & \ (False & \mapsto & bool), \\
 & & \ (Zero & \mapsto & nat), \\
 & & \ (Succ & \mapsto & nat \rightarrow nat), \\
 & & \ (plus & \mapsto & nat \rightarrow nat \rightarrow nat), \\
 & & \ (Nil & \mapsto & \forall\alpha.(list\ \alpha)), \\
 & & \ (Cons & \mapsto & \forall\alpha.\alpha \rightarrow (list\ \alpha) \rightarrow (list\ \alpha)), \\
 & & \ (is_empty & \mapsto & arbre \rightarrow bool), \\
 & & \ (MkPair & \mapsto & \forall\alpha\ \forall\beta.\alpha \rightarrow \beta \rightarrow (pair\ \alpha\ \beta)), \\
 & & \ (left & \mapsto & \forall\alpha\ \forall\beta.(pair\ \alpha\ \beta) \rightarrow \alpha), \\
 & & \ (right & \mapsto & \forall\alpha\ \forall\beta.(pair\ \alpha\ \beta) \rightarrow \beta)\}
\end{array}$$

FIG. 5.3 – Exemple d'une signature de types

nous noterons Γ, permet d'associer à chaque symbole d'une signature d'expressions d'une spécification JSL — les symboles de fonction et de constructeur, son type. C'est la version ensembliste de la fonction *type* de $\Sigma_{\mathcal{T}}$. Le deuxième, le contexte que nous noterons Δ, est la donnée des variables de termes, associées à leur type, présentes dans la règle de réécriture qui contient l'expression à vérifier. Nous noterons les éléments de ces deux ensembles par $(s : t)$ où :

- s est soit un symbole de $\mathcal{C} \cup \mathcal{D}$ soit un élément de $\mathcal{V}$;
- t est le type associé à s.

Nous avons, pour une règle de réécriture r, $\mathsf{dom}(\Delta) = \mathsf{dom}(r)$. L'information du type des variables est obtenue par inférence de type à partir de r et de S_{pec}.

Jugement de bon typage La propriété de bon typage est alors présentée comme un prédicat $\vdash$ qui mettra en relation un énoncé, noté $(e : t)$, avec un environnement et, éventuellement, un contexte. Classiquement $(e : t)$ a pour sens "e a pour type t", et $\Gamma, \Delta \vdash e : t$, appelé jugement de type, dénote "e a pour type t dans l'environnement Γ et dans le contexte Δ". La définition du prédicat $\vdash$ établit les conditions sous lesquelles un tel jugement est vérifié. Tout l'objet de la vérification du typage consiste à vérifier, pour un jugement donné, le statut de ces conditions, et donc de confirmer ou d'infirmer ce jugement.

La définition de $\vdash$ ne pose pas de problème particulier. Exceptée, peut être, la présence des synonymes de type qui oblige à ce que la vérification d'un jugement $\Gamma, \Delta \vdash e : t$ soit faite modulo la conversion $=_\delta$.

Cette conversion est définie de façon suivante :

Définition 10 (Réduction des types)

- *Rappelons que $\mathcal{T}_s$ est l'ensemble des symboles de synonymes de type. On définit $\rightarrow_\delta$ la relation binaire sur les expressions de type par :*

$$\forall n \geq 0, \forall t \in \mathcal{T}_s.\ (synonym\ t) = (\forall \alpha_1 \cdots \forall \alpha_n.s) \Rightarrow\ t \rightarrow_\delta s$$

- *De façon standard, on définit $\triangleright_\delta$ comme étant la relation correspondant à la réécriture d'une sous-expression de type par $\rightarrow_\delta$. On désigne par $\triangleright_\delta^*$ et $=_\delta$ les clôtures réflexive transitive, réflexive transitive symétrique de $\triangleright_\delta$ et $\downarrow_\delta$ la forme normale.*

En reprenant l'exemple de spécification JSL de la section 5.1.1, où nous introduisons la fonction *synonym* suivante :

$$\begin{array}{rcllcl} synonym & = & \{(list\_nat & \mapsto & (list\ nat)), \\ & & \ (sequence & \mapsto & \forall \alpha.(list\ \alpha))\} \end{array}$$

nous déduisons le système suivant : $\{list\_nat \rightarrow_\delta (list\ nat),\ sequence \rightarrow_\delta (list\ \alpha)\}$.

Terminaison et confluence Tout d'abord, $\rightarrow_\delta$ induit un système de réécriture linéaire gauche par construction. Puis, les contraintes de bonne formation assurent que, pour chaque symbole de synonyme de type, il n'existe qu'une seule règle la réduisant. Puisque la définition de $\rightarrow_\delta$ ne contient pas de paire critique, le système est orthogonal, dès lors confluent (voir [98]). De plus, ces mêmes contraintes assurent l'unicité de chaque symbole, ce qui a pour conséquence directe qu'il n'y a pas de cycle dans le système de réécriture. Celui-ci est donc terminant. Pour une expression de type t, nous noterons donc, puisque la confluence et la terminaison nous le garantissent, par $t \downarrow_\delta$ la forme normale unique de t par $\triangleright_\delta^*$.

Comme nous l'avons déjà mentionné, la définition du jugement de bon typage d'un terme est tout à fait classique. On utilise la notation $\Gamma \vdash r$ pour le jugement de bon typage d'une règle et qui signifie la règle de réécriture r est bien typée dans l'environnement Γ, c'est-à-dire que toutes les expressions la formant sont bien typées. La vérification de typage d'une règle introduit la contrainte suivante : le membre gauche et le membre droit d'une condition ont même type. Cette contrainte est aussi imposée aux expressions représentant une égalité.

Définition 11 (Bon typage d'une règle de réécriture)

$$\frac{\begin{array}{cc} (f : \forall \alpha_0 \ldots \alpha_m . \sigma_1 \rightarrow \cdots \rightarrow \sigma_p \rightarrow \sigma) \in \Gamma & m \geq 0 \\ \Gamma, \Delta \vdash f\ x_1 \cdots x_p : \sigma & \Gamma, \Delta \vdash d : \sigma \\ \Gamma, \Delta \vdash l_1 : \tau_1 \quad \Gamma, \Delta \vdash r_1 : \tau_1 \quad \ldots & \Gamma, \Delta \vdash l_n : \tau_n \quad \Gamma, \Delta \vdash r_n : \tau_n \end{array}}{\Gamma \vdash (l_1 \twoheadrightarrow r_1,\ \ldots\ ,\ l_p \twoheadrightarrow r_p \Rightarrow f\ x_1 \cdots x_p \rightarrow d)}$$

Pour qu'une règle $(l_1 \twoheadrightarrow r_1,\ \ldots\ ,\ l_p \twoheadrightarrow r_p \Rightarrow f\ x_1 \cdots x_p \rightarrow d)$ soit bien typée, il faut que f soit un symbole de fonction d'arité p et ait un type de la forme $\forall \alpha_0 \ldots \alpha_m . \sigma_1 \rightarrow \cdots \rightarrow \sigma_p \rightarrow \sigma$. Dans ce cas, il faut aussi que dans chaque condition, le membre gauche ait le même type que le membre droit, avec le cas particulier de la condition principale où ce type doit être le type de retour de f.

Partialité et confluence Le typage des règles de réécriture et des fonctions JSL ne force ni la totalité ni la confluence des fonctions. Néanmoins, nous verrons, section 5.2.4, que nous avons implémenté des mécanismes permettant d'exhiber les fonctions ne vérifiant pas ces critères ainsi que des mécanismes permettant de modifier ces comportements.

Définition 12 (Bon typage d'une expression)

$$\mathit{Tbase}_1\frac{(x:\sigma)\in\Gamma}{\Gamma\vdash x:\sigma}\qquad \mathit{Tbase}_2\frac{(f:\forall\alpha_1\ldots\alpha_m.\sigma_1\to\cdots\to\sigma_n\to\sigma)\in\Gamma}{\Gamma\vdash f:\forall\alpha_1\ldots\alpha_m.\sigma_1\to\cdots\to\sigma_n\to\sigma}$$

$$\mathit{Teq}\frac{\Gamma\vdash t_1:\sigma\qquad\Gamma\vdash t_2:\sigma}{\Gamma\vdash(t_1==t_2):\mathit{bool}}\qquad \mathit{Texpr}\frac{\begin{array}{c}\Gamma\vdash f:\sigma_1\to\cdots\to\sigma_n\to\sigma\\ \Gamma\vdash t_1:\sigma_1\quad\ldots\quad\Gamma\vdash t_n:\sigma_n\end{array}}{\Gamma\vdash f\ t_1\ldots t_n:\sigma}$$

$$\mathit{Tinst}\frac{\Gamma\vdash f:\forall\alpha_1\ldots\alpha_m.\sigma_1\to\cdots\to\sigma_n\to\sigma\qquad f\in\mathcal{C}\cup\mathcal{D}\qquad m\geq 0}{\Gamma\vdash f:(\sigma_1\to\cdots\to\sigma_n\to\sigma)\{\alpha_1:=\tau_1,\ldots,\alpha_m:=\tau_m\}}$$

$$\mathit{Tconv}\frac{\Gamma\vdash t:\sigma'\qquad\sigma=_\delta\sigma'}{\Gamma\vdash t:\sigma}$$

Les règles $Tbase_1$ et $Tbase_2$ sont les axiomes du système de typage. La règle $Texpr$ exprime qu'une expression $f\ t_1 \ldots t_n$ est bien typée avec le type σ si chacun des paramètres est bien typé et si les types des paramètres associés au type σ forment bien le type de la fonction f. La règle Teq impose la contrainte qu'une égalité est bien typée par $Bool$ si les deux membres de l'égalité sont bien typés par le même type. La règle $Tinst$ établit que si un symbole de fonction ou de constructeur f a pour type $\forall\alpha_1 \ldots \alpha_m.\sigma_1 \to \cdots \to \sigma_n \to \sigma$ alors celui-ci peut être instancié. La règle $Tconv$ introduit le bon typage par conversion : une expression t est bien typée par un type σ si l'on sait la typer avec un type $\delta-$convertible.

5.1.4 L'outil JaKarTa

Dans le cadre d'une implémentation du langage JSL, une spécification JSL est représentée par une séquence de définitions permettant la construction incrémentale des ensembles Σ et $\Sigma_\mathcal{T}$. Nous appellerons cette séquence *spécification JaKarTa*. Intuitivement, une spécification JaKarTa est une spécification JSL en syntaxe concrète où toutes les contraintes sont vérifiées.

La vérification d'une spécification JaKarTa est divisée en deux parties, la *vérification de la bonne formation* et la *vérification du typage*. Une définition JaKarTa est ainsi acceptée si elle passe avec succès ces deux phases de vérification.

Dans cette section, nous présentons l'outil JaKarTa par l'introduction d'un exemple de spécification simple (figures 5.4 et 5.5) que nous reprendrons dans la section 5.2 pour illustrer les mécanismes d'abstraction.

Les types sont introduits par les mots clés **`data`**, pour les types inductifs, et **`type`** pour les autres.

```
(***********************)
(* structure de donnees *)
(***********************)

data nat = Zero | Succ  nat.

data bool = True | False.

type boolean = bool.

data list = Nil | Cons  'a (list 'a).

record pair = {left  : 'a ; right : 'b}.

data err = Val 'a | Err.

type value.

type rationnel.

type arbre 'a.

type sequence = (list 'a).

type list_nat = (list nat).

data tvalue  = Short value | Long  value | Ref value.
```

FIG. 5.4 – EXEMPLE D'UNE SPÉCIFICATION JAKARTA

JaKarTa met en place quelques facilités syntaxiques. Remarquons ainsi qu'il n'est pas nécessaire de donner le type complet dans la définition d'un constructeur. JaKarTa se charge de le compléter. La syntaxe pour les variables de type est celle d'OCaml : les identificateurs sont préfixés par une apostrophe. Remarquons aussi la déclaration de la ligne 13 fig. 5.4 : une syntaxe spéciale est mise en place pour les types enregistrement qui sont traduit dans JSL, à l'image du système Coq, comme des types inductifs à un seul constructeur. Les notations habituelles pour les éléments de ces types sont disponibles (`x.left`, `{left=y;right=z}`, `{x` **`with`** `left=x.right}` en sont des exemples). Le type `tvalue` (ligne 27) introduit un ensemble de valeurs typées. Les types considérés sont les références, les valeurs de "petite taille" et les valeurs de "grande taille".

Dans la figure introduisant des exemples de fonctions (fig. 5.5) notons le sucre syntaxique consistant à laisser des motifs apparaître dans le membre gauche de la condition principale (lignes 6 et 7). Notons aussi la possibilité de nommer les règles (ligne 6). Le mot clé **`definition`** est une abréviation pour l'introduction pour les fonctions constantes (ligne 15).

La définition de la ligne 17 déclare une fonction qui additionne le contenu de deux valeurs de type `tvalue` si celles-ci sont de même nature (dans ce cas, si elles ont même constructeur). La valeur d'erreur `Err` est retournée le cas contraire.

Notons que l'addition de références n'est pas possible si l'une des références considérées est nulle (`Err` est donc la valeur de retour). Nous utiliserons cette fonction pour illustrer l'action du JTK.

```
(**********************)
(*      fonctions     *)
(**********************)

function plus : nat → nat → nat :=
<base_case> ⇒ (plus x Zero) → x ;
            ⇒ (plus x (Succ z)) → (Succ (plus x z)).

variable is_null : value → bool.

variable vplus :  value →  value  → value.

variable vdefault : value.

definition tvaluedefault :  tvalue := (Ref vdefault).

function add : tvalue → tvalue → err tvalue :=

   n → Short vn, m → Short vm,
   (vplus vn vm) → v            ⇒ add n m → Val (Short v);

   n → Long vn,  m → Long vm,
   (vplus vn vm) → v            ⇒ add n m → Val (Long v);

   n → Ref vn,   m → Ref vm,
   is_null vn → False, is_null vm → False,
   (vplus vn vm) → v            ⇒ add n m → Val (Ref v);

   n → Ref vn,   m → Ref vm,
   is_null vn → True, is_null vm → True
                                 ⇒ add n m → Err;

   n → Ref vn,   m → Ref vm,
   is_null vn → True, is_null vm → False
                                 ⇒ add n m → Err;

   n → Ref vn,   m → Ref vm,
   is_null vn → False, is_null vm → True
                                 ⇒ add n m → Err;

   n → Short vn, m → Long vm     ⇒ add n m → Err;

   n → Short vn, m → Ref vm      ⇒ add n m → Err;

   n → Long vn,  m → Ref  vm     ⇒ add n m → Err;

   n → Long vn,  m → Short vm    ⇒ add n m → Err;

   n → Ref vn,   m → Long vm     ⇒ add n m → Err;

   n → Ref vn,   m → Short vm    ⇒ add n m → Err.
```

FIG. 5.5 – EXEMPLE D'UNE SPÉCIFICATION JAKARTA (SUITE)

5.2 JTK : support à l'abstraction

La plus importante des fonctionnalités de JaKarTa est le support à la construction d'abstractions offert par le JTK. Comme nous l'avons souligné dans l'introduction, le format choisi pour les règles JSL facilite (1) la manipulation des spécifications JSL (2) le développement de procédures de transformation pour ce propos. Nous exposerons dans ce qui suit et par le biais d'un exemple simple comment le JTK accomplit de telles transformations.

5.2.1 Une présentation par l'exemple

Soit `S` la spécification, que nous qualifierons de *concrète*, présentée dans les figures 5.4 et 5.5, et que nous cherchons à abstraire. Supposons que l'on veuille réaliser l'abstraction qui consiste à oublier les valeurs et ne conserver que les types. Cette variante de l'abstraction de type agit sur le type `tvalue`. Concrètement nous désirons transformer les valeurs de `tvalue` en des valeurs du type `tvalue_abs`.

```
data tvalue_abs = Short_abs  | Long_abs | Ref_abs.
```

La fonction `abstract_tvalue` est la fonction d'abstraction qui transforme les valeurs `tvalue` en valeurs `tvalue_abs`. Celle-ci respecte les contraintes propres aux fonctions d'abstraction :

```
function abstract_tvalue : tvalue → tvalue_abs :=
  ⇒ abstract_tvalue (Short n) → Short_abs;
  ⇒ abstract_tvalue (Long  n) → Long_abs;
  ⇒ abstract_tvalue (Ref   n) → Ref_abs.
```

La synthèse de la spécification *abstraite* $\hat{\texttt{S}}$ est dirigée par l'interprétation, par le JTK, d'un ensemble de commandes regroupées par l'utilisateur dans ce que l'on appelle un *script d'abstraction*. Celui-ci indique dans son préambule, quelle est la fonction `f` à abstraire et comment sera nommé le résultat $\hat{\texttt{f}}$. Toutes les fonctions présentes dans le graphe de dépendance de `f` seront sujettes à ce même processus d'abstraction. Le préambule, que l'on appelle *spécification d'abstraction*, introduit l'ensemble des fonctions d'abstraction à utiliser (une pour chaque type que l'on veut abstraire). Cette première partie du script fournit toutes les informations nécessaires au processus de traduction syntaxique qui est automatique. Il est cependant souvent nécessaire de modérer ce processus. C'est dans la seconde partie du script que l'utilisateur place les commandes spécifiant les modifications qu'il désire apporter. Nous verrons par la suite qu'il y a trois catégories de commandes, les commandes de suppression, les commandes de substitution et les commandes de transformation. La figure 5.6 est le script mettant en place l'abstraction de type désirée. Les lignes 4 et 5 forment le préambule, et indiquent au JTK que la fonction `add` est à abstraire en `typed_add` et que l'abstraction désirée est définie par la fonction `abstract_tvalue`. La ligne 9 indique dans quels fichiers seront placés le résultat de l'abstraction et la trace des opérations effectuées. La ligne 8 introduit l'unique commande de ce script et qui demande à ce que l'abstraction de la fonction `add` soit déterministe.

Décrivons maintenant comment est générée la fonction `typed_add`. L'abstraction est conduite en trois phases :

```
(* Preambule *)

build typed_add from  add
abstraction functions abstract_tvalue

(* commandes *)
determine add
into typed_add_abs  log abs_log.
```

FIG. 5.6 – EXEMPLE D'UN SCRIPT D'ABSTRACTION

phase de traduction syntaxique : les fonctions d'abstraction induisent deux types de transformation sur les fonctions JSL de la spécification considérée. L'une sur le type de ces fonctions, l'autre sur leurs règles. La transformation des types est une simple substitution construite à partir de l'ensemble des types des fonctions d'abstraction. Ici nous n'avons à prendre en compte que $tvalue \rightarrow tvalue\_abs$. La substitution que nous considérons ici est donc $\sigma = [tvalue := tvalue\_abs]$. Ainsi le type `tvalue →tvalue →(err tvalue)` de `add` est réduit par l'application de σ vers `tvalue_abs →tvalue_abs →(err tvalue_abs)`.

La deuxième partie de la traduction s'attache à réduire les règles d'une fonction. Il s'agit d'une étape de réécriture paramétrée par la définition de `abstract_tvalue`. Le système de réécriture est défini par les règles suivantes :

$$\{(Short\, n \rightarrow Short\_abs);\ (Long\, n \rightarrow Long\_abs);\ (Ref\ n \rightarrow Ref\_abs)\}$$

L'application de ce système sur la règle de la ligne 27 de la définition de `add` (figure 5.5) donne

```
n → Ref_abs,  m → Ref_abs, is_null vn → False,
is_null vm → False, (vplus vn vm) → v
          ⇒ add n m → Val Ref_abs;
```

Remarquons que les fonctions `is_null` et `vplus` ne sont pas modifiées. En effet elles ne manipulent pas de valeurs qui sont sujettes à l'abstraction.

C'est aussi à ce moment que sont prises en compte les commandes de suppression et de substitution. Dans notre exemple, nous n'en avons pas ;

phase de nettoyage : après traduction, il faut vérifier si le résultat est conforme au format JSL. Le JTK lance, à cet effet, les deux modules de vérification de contraintes : la vérification de bonne formation et la vérification du typage. Toute expression violant les contraintes JSL est supprimée. Il est cependant possible de remédier à certaines violations du typage par le biais de commandes de coercions (voir section 5.2.6).

Dans le cas de la suppression d'une expression, il est nécessaire de propager cette suppression afin de conserver la cohérence de l'ensemble de la définition. Ainsi la suppression d'un membre d'une condition implique la suppression de la condition (une condition doit avoir un membre gauche et un membre droit), de même si la conclusion d'une règle viole une contrainte, c'est toute la règle qui est supprimée car il ne peut y avoir de règle sans conclusion.

Dans notre exemple, le membre gauche de la troisième condition (`is_null vn` →`False`) contient une occurrence de la variable `vn` qui n'a pas été préalablement introduite (la version concrète de la règle introduisait `vn` dans la première condition). De façon tout à fait semblable les quatrième et cinquième conditions violent les contraintes de bonne formation des règles JSL. La règle devient donc :

`n` → `Ref_abs`, `m` → `Ref_abs` ⇒ `add n m` → `Val Ref_abs;`

Remarquons que les règles lignes 31, 35 et 39 de la définition de `add` sont toutes les trois traduites vers :

`n` → `Ref_abs`, `m` → `Ref_abs` ⇒ `add n m` → `Err;`

Soulignons que ces deux dernières règles induisent une fonction non-déterministe ;

phase de transformation : le JTK vérifie à ce moment les règles de chaque fonction générée. Le JTK optimise le résultat en supprimant les règles dupliquées et en interprétant une catégorie de commandes de suppression traitant d'un cas spécial de duplication de règles (la commande **`reject`**, voir figure 5.9). Dans notre exemple, la cinquième et la sixième règles, qui sont des duplications de la quatrième règle, sont retirées.

Vient ensuite une phase de test supplémentaire : pour chaque fonction, le JTK indique dans un premier temps si celle-ci est partielle et dans un deuxième temps si celle-ci est non-déterministe. Une fonction est non-déterministe lorsqu'il existe au moins deux règles telles que certaines valeurs de leurs variables ne permettent pas de les différentier. On dit alors que ces deux règles se recouvrent. Si la fonction est non-déterministe, la liste des règles se recouvrant, couplées aux valeurs (sous la forme d'une substitution) exhibant ces recouvrements, est calculée et affichée sous la forme d'un message de *warning*.

C'est aussi à ce moment que sont prises en compte les commandes de transformation. Dans notre exemple, si on omet la commande ligne 8 de la figure 5.6, le JTK nous informe que la troisième règle et la quatrième règle se recouvrent avec la substitution vide. C'est ce comportement indésirable que la commande de transformation **`determine`** (ligne 8) permet d'éviter. En particulier, il est aisé de palier le recouvrement de deux règles qui ont exactement le même corps (c'est-à-dire qui ne diffèrent que par la conclusion). C'est le cas de l'abstraction de la fonction `add`. La solution consiste à rassembler les différentes conclusions dans une liste, et à transformer le type de retour $\hat{\texttt{t}}$ de la fonction considérée en (*list* $\hat{\texttt{t}}$). Il est, bien sûr, nécessaire de propager cette modification aux fonctions appelantes. Le résultat final de l'abstraction est donné dans la figure 5.7. Remarquons ainsi que les conclusions des deux règles conflictuelles sont regroupées dans une liste (ligne 10).

Des exemples complets d'abstraction calculées par le JTK sont exposés chapitre 6. Dans les sections suivantes nous nous attachons à décrire le processus

```
function typed_add : tvalue_abs → tvalue_abs → (list (err tvalue_abs)) :=

 n → Short_abs, m → Short_abs  ⇒
          (typed_add n m) →  (Cons (Val Short_abs) Nil);

 n → Long_abs,  m → Long_abs   ⇒
          (typed_add n m) →  (Cons (Val Long_abs) Nil);

 n → Ref_abs,   m → Ref_abs    ⇒
          (typed_add n m) →  (Cons (Val Ref_abs) (Cons Err Nil));

 n → Ref_abs,   m → Long_abs   ⇒
          (typed_add n m) →  (Cons Err Nil);

 n → Ref_abs,   m → Short_abs  ⇒
          (typed_add n m) →  (Cons Err Nil);

 n → Long_abs,  m → Short_abs  ⇒
          (typed_add n m) →  (Cons Err Nil);

 n → Long_abs,  m → Ref_abs    ⇒
          (typed_add n m) →  (Cons Err Nil);

 n → Short_abs, m → Long_abs   ⇒
          (typed_add n m) →  (Cons Err Nil);

 n → Short_abs, m → Ref_abs    ⇒
          (typed_add n m) →  (Cons Err Nil).
```

Fig. 5.7 – Résultat final de l'abstraction

d'abstraction par la revue en détail des différentes phases du calcul pour ensuite introduire le langage de commandes ainsi que l'interprétation de celles-ci au sein du processus de traduction.

5.2.2 Le processus de traduction syntaxique

Nous considérons ici la description formelle du moteur de traduction du JTK. Celui-ci transforme syntaxiquement une spécification S dite *concrète* en une une spécification $\hat{\mathtt{S}}$ dite *abstraite*.

Dans cette section, nous nous attacherons à décrire graduellement le procédé de traduction syntaxique en commençant par la configuration la plus simple puis en raffinant vers le modèle complet. Nous verrons ensuite, dans la section 5.2.5, comment l'interprétation des commandes du script d'abstraction permet de modifier le comportement du processus de traduction.

Processus d'abstraction par substitution de motifs

Pour commencer, décrivons la configuration de base. On associe à chaque type T un type abstrait $\hat{\mathtt{T}}$ et une fonction d'abstraction $\alpha : T \to \hat{\mathtt{T}}$ décrite par des règles JSL de la forme :

$$\mathtt{x} \!\to\! \mathtt{r} \;\Rightarrow\; \alpha\ \mathtt{x} \to \hat{\mathtt{r}}$$

où r et $\hat{\mathtt{r}}$ sont nécessairement des motifs.

Nous appellerons *spécification d'abstraction*, notée $\mathcal{A}$, l'ensemble de ces associations.

La définition d'une spécification d'abstraction peut se révéler problématique. Nous verrons dans la section 5.2.2 que le choix des types à abstraire et des fonctions d'abstractions n'est pas anodin.

Notations Dans les définitions qui suivent, nous sous-entendrons l'existence de la spécification JSL `S` sur laquelle a lieu l'abstraction et de la spécification d'abstraction $\mathcal{A}$. $\mathcal{A}(\tau_c, \tau_a)$ pour deux types τ_c et τ_a donnés, est la fonction partielle qui retourne la fonction d'abstraction de $\mathcal{A}$ dont la signature est $\tau_c \rightarrow \tau_a$. Nous nous permettrons d'utiliser la notation $(\tau_c, \tau_a) \in \mathcal{A}$ (respectivement $(\tau_c, \tau_a) \notin \mathcal{A}$) pour dénoter l'existence (respectivement la non-existence) dans $\mathcal{A}$ d'une fonction d'abstraction de τ_c vers τ_a. Plus généralement, la notation $\tau \in \mathcal{A}$ (ou $\tau \notin \mathcal{A}$) dénotera l'existence (ou non) d'une fonction d'abstraction pour le type τ dans la spécification $\mathcal{A}$. Soit τ le type d'un constructeur c, nous noterons par $\mathcal{A}(c)$ la fonction d'abstraction, élément de $\mathcal{A}$, qui abstrait les éléments de τ. Finalement, nous noterons $\hat{f}$ le symbole résultant de l'abstraction du symbole f — soulignons que $\hat{f}$ peut être f dans le cas où la fonction n'est pas sujette à l'abstraction.

Les définitions suivantes introduisent les éléments nécessaires à la définition de l'abstraction $\hat{\texttt{f}}$ d'une fonction `f`. Pour ce faire, le processus d'abstraction devra être en mesure de traduire deux catégories de données : les types et les règles.

Définition 13 (Abstraction de types)

Soit σ la substitution sur $\mathcal{E}_\mathcal{T}$ définie par :

$$\forall (\tau, \tau_\alpha) \in \mathcal{A}.\ \tau := \tau_\alpha$$

Alors l'opérateur d'abstraction de type, noté $\lceil . \rceil : \mathcal{E}_\mathcal{T} \rightarrow \mathcal{E}_\mathcal{T}$ est défini par

$$\lceil T \rceil \rightsquigarrow \sigma(T)$$

La section 5.2.1 présente un exemple simple d'abstraction de type. Pour ce qui est de l'abstraction d'expressions nous avons :

Définition 14 (Abstraction d'expressions - version 1)

$\lceil . \rceil : \mathcal{E} \rightarrow \mathcal{E}$, *l'opérateur d'abstraction d'expression est récursivement défini par :*

$$\mathit{Exp}_v \frac{x \in \mathcal{V}}{\lceil x \rceil \leadsto x} \qquad \mathit{Exp}_{eq} \frac{}{\lceil e_1 == e_2 \rceil \leadsto \lceil e_1 \rceil == \lceil e_2 \rceil}$$

$$\mathit{Exp}_f \frac{f \in \mathcal{D}}{\lceil (f\ t_1 \ldots t_n) \rceil \leadsto \hat{f}\ \lceil t_1 \rceil \ldots \lceil t_n \rceil} \qquad \mathit{Exp}_{m1} \frac{c \in \mathcal{C} \quad \neg\exists\alpha.\ \alpha = \mathcal{A}(c)}{\lceil c\ t_1 \ldots t_n \rceil \leadsto c\ \lceil t_1 \rceil\ \ldots\ \lceil t_n \rceil}$$

$$\mathit{Exp}_{m2} \frac{\begin{array}{c} c \in \mathcal{C} \\ \exists\alpha.\ \alpha = \mathcal{A}(c) \wedge\ (x \twoheadrightarrow (c\ x_1 \ldots x_n)\ \Rightarrow (\alpha\ x) \rightarrow (c'\ y_1 \ldots y_p)) \in \alpha \\ \sigma = [x_1 \leftarrow t_1\ \ldots\ x_n \leftarrow t_n] \end{array}}{\lceil c\ t_1 \ldots t_n \rceil \leadsto (c'\ \sigma(y_1)\ \ldots\ \sigma(y_p))}$$

Dans ce cadre simple, l'abstraction d'une variable est l'identité. La différence se situe au typage, la variable acquiert la version abstraite de son type original. L'abstraction d'une égalité et d'un appel de fonction est évidente. Celle d'un motif requiert la recherche dans $\mathcal{A}$ d'une fonction d'abstraction adéquate. Si cette fonction n'existe pas alors on se contente d'abstraire les sous-motifs. S'il existe une fonction d'abstraction alors la substitution correspondant à la traduction du motif vers sa version abstraite est construite et appliquée.

Processus d'abstraction par réécriture de motifs

De fait, le format des règles de fonction d'abstraction que nous avons présentée est simple, mais trop restrictif. Nous devons pouvoir considérer les cas où $\hat{\mathtt{r}}$ n'est pas un motif mais plus généralement un terme t dont l'évaluation conduit à un motif :

$$\mathtt{x} \rightarrow \mathtt{r} \ \Rightarrow\ \alpha\ \mathtt{x} \rightarrow t$$

Par là nous entendons que les fonctions d'abstraction peuvent appeler d'autres fonctions. Celles-ci peuvent être, par exemple, des fonctions d'abstraction pour les champs d'un constructeur ou encore des fonctions auxiliaires.

Un exemple concret peut être :

```
data nat = Zero | Succ  nat.

data bool = True | False.

data T = C nat .

data Ta = Ca bool.

function abstract_nat : nat → bool :=
   ⇒  abstract_nat  Zero     → True ;
   ⇒  abstract_nat  (Succ m) → False .

function abstract_T : T → Ta :=
y → C z ⇒ abstract_T y →  Ca (abstract_nat z).
```

Dans cet exemple nous désirons abstraire le type `T` en `Ta`. Or cela implique que des entiers soient abstraits en booléens. La fonction d'abstraction `abstract_T` fait donc appel à `abstract_nat`. Il aurait été difficile d'exprimer une telle abstraction dans le cadre de l'approche précédente.

Le sens qu'il faut attribuer à de telles fonctions d'abstraction est le suivant : chaque fonction d'abstraction $\alpha : \tau \rightarrow \hat{\tau}$ induit un système de réécriture, noté $\mathcal{AR}(\alpha)$, sur les motifs — et nous nous restreignons à ce niveau à la réécriture inconditionnelle, composé par toutes les règles des fonctions du graphe de dépendance de α. Alors, l'abstraction de tout motif m de type τ est le motif $m \downarrow_{\mathcal{AR}(\alpha)}$ de type $\hat{\tau}$ obtenu par réduction par $\mathcal{AR}(\alpha)$.

Terminaison et confluence Nous assumerons ici que le système de réécriture sous-jacent à l'abstraction est terminant et confluent, ce qui nous autorise à parler de forme normale. Cependant ce système est construit à partir des données fournies par l'utilisateur. La responsabilité du respect des critères de terminaison et de confluence lui est donc laissée. Néanmoins, abstraction faite de la terminaison des fonctions d'abstraction, ces critères sont simples à respecter :

- il ne faut pas définir deux fonctions d'abstraction pour le même type. La version actuelle de JaKarTa ne prendra, de toute façon, en compte que la première des définitions ;
- il faut bannir les fonctions d'abstraction et fonctions auxiliaires instaurant un cycle, c'est-à-dire, pour tout $n \geq 2$, des fonctions $\alpha_1, \ldots, \alpha_n$ telles que $\alpha_1 : A_1 \rightarrow A_2, \ldots, \alpha_n : A_n \rightarrow A_1$.

Définition 15 (Abstraction d'expressions - version 2)
$\lceil . \rceil : \mathcal{E} \rightarrow \mathcal{E}$, l'opérateur d'abstraction d'expression est récursivement défini par :

$$Exp_v \frac{x \in \mathcal{V}}{\lceil x \rceil \rightsquigarrow x} \qquad Exp_{eq} \frac{}{\lceil e_1 == e_2 \rceil \rightsquigarrow \lceil e_1 \rceil == \lceil e_2 \rceil}$$

$$Exp_f \frac{f \in \mathcal{D}}{\lceil (f\ t_1 \ldots t_n) \rceil \rightsquigarrow \hat{f}\ \lceil t_1 \rceil \ldots \lceil t_n \rceil} \qquad Exp_{m1} \frac{c \in \mathcal{C} \qquad \neg\exists \alpha.\ \alpha = \mathcal{A}(c)}{\lceil c\ t_1 \ldots t_n \rceil \rightsquigarrow c\ \lceil t_1 \rceil\ \ldots\ \lceil t_n \rceil}$$

$$Exp_{m2} \frac{c \in \mathcal{C} \qquad \exists \alpha.\ \alpha = \mathcal{A}(c)}{\lceil c\ t_1 \ldots t_n \rceil \rightsquigarrow (c\ t_1 \ldots t_n) \downarrow_{\mathcal{AR}(\alpha)}}$$

Processus d'abstraction par réécriture avec opérateur d'abstraction de variable

Malgré son apparente simplicité, l'abstraction de variable soulève quelques subtilités. En particulier parce que cette opération est sensible à la position de la variable considérée dans la règle de réécriture. Considérons le code JaKarTa suivant.

```
data nat = Zero | Succ  nat.

data bool = True | False.

data T = C nat.

data Ta = Ca bool.

function h : nat → T:=
   ⇒ h n →   (C n).

function natbool : nat → bool :=
y → z   ⇒  natbool  y → True.

function abstract_T : T → Ta :=
y → C z ⇒ abstract_T y →  Ca (natbool z).

build abs_h from h  abstraction functions  abstract_T.
```

Nous désirons abstraire la fonction `h` avec l'abstraction transformant toute valeur de `T` en valeur de `Ta`. Remarquons que :

- la fonction d'abstraction `abstract_T` utilise `natbool` comme fonction auxiliaire et que cette fonction n'est pas une fonction d'abstraction ;
- nous construisons le motif `(C n)` de type `T` dans la fonction `h` à partir de son paramètre `n` de type `nat`.

La difficulté réside dans l'abstraction de la variable `n` qui a une occurrence aussi bien dans le terme `(C n)` que dans le terme `h n`. En effet, l'*abstraction naïve* de ces deux termes, c'est-à-dire l'opération considérant l'abstraction d'une variable comme l'identité, donnerait : `⇒abs_h n →Ca n.` Or, dans `(Ca n)`, l'occurrence de `n` a pour type `bool` alors que `nat` est le type de l'occurrence dans `h n`.

La solution adoptée est de donner préférence à la première occurrence (celle de `(abs_h n)` en prenant compte de l'ordre sur les positions des termes dans une règle). Le type abstrait de `n` est donc fixé à `nat`. Si une abstraction naïve donne à tout autre occurrence de `n` un type différent alors l'abstraction est gelée, c'est-à-dire que l'application de la fonction d'abstraction sur la variable n'est pas interprétée. Dans notre exemple l'abstraction de `n` dans `(C n)` par la fonction d'abstraction auxiliaire `natbool` est gelée en `(natbool n)` au lieu d'être évaluée en `n`. Remarquons que dans ce cas `(natbool n)` a bien le type escompté, `bool`. Le résultat final est donc :

```
function  abs_h  : nat → nat :=
    ⇒ (abs_h n) → (Ca (natbool n)).
```

Remarquons que les commandes de coercion fournissent un support adéquat à la résolution des subtilités de l'abstraction de variables. Avec V_{conv}, nous avons préféré résoudre le problème de typage de variable au sein du processus même d'abstraction, car ce problème est dû, non pas à une faute de spécification de l'utilisateur mais bien à une subtilité du processus d'abstraction dont l'utilisateur n'est pas responsable. Il n'est donc pas raisonnable de lui demander, par une commande de coercion adéquate, de palier ce problème.

Soit ν l'opérateur d'abstraction de variables réalisant une telle abstraction. La définition d'un tel opérateur ne pose que des problèmes mineurs et repose sur les notions d'occurrence et de position des variables ainsi que sur la notion d'inférence de type. Nous noterons par $t \downarrow_{\mathcal{AR}(\alpha),\nu}$ la forme normale de t par $\mathcal{AR}(\alpha)$ étendu avec la gestion ν des variables. Finalement nous avons donc :

Définition 16 (Abstraction d'expressions - version 3)
$\lceil . \rceil : \mathcal{E} \rightarrow \mathcal{E}$, l'opérateur d'abstraction d'expression est récursivement défini par :

$$Exp_v \frac{x \in \mathcal{V}}{\lceil x \rceil \rightsquigarrow \nu(x)} \qquad Exp_{eq} \frac{}{\lceil e_1 == e_2 \rceil \rightsquigarrow \lceil e_1 \rceil == \lceil e_2 \rceil}$$

$$Exp_f \frac{f \in \mathcal{D}}{\lceil (f\ t_1 \dots t_n) \rceil \rightsquigarrow \hat{f}\ \lceil t_1 \rceil \dots \lceil t_n \rceil} \qquad Exp_{m1} \frac{c \in \mathcal{C} \qquad \neg \exists \alpha.\ \alpha = \mathcal{A}(c)}{\lceil c\ t_1 \dots t_n \rceil \rightsquigarrow c\ \lceil t_1 \rceil\ \dots\ \lceil t_n \rceil}$$

$$Exp_{m2} \frac{c \in \mathcal{C} \qquad \exists \alpha.\ \alpha = \mathcal{A}(c)}{\lceil c\ t_1 \dots t_n \rceil \rightsquigarrow (c\ t_1 \dots t_n) \downarrow_{\mathcal{AR}(\alpha),\nu}}$$

En pratique, les cas d'abstraction d'expressions nécessitant l'intervention de ν sont rares. Mais nous en avons rencontré, essentiellement dans des fonctions de conversion (qui construisent donc des motifs du type cible). Dans le cadre des abstractions que nous détaillons au chapitre 6, certaines fonctions auxiliaires aux instructions de manipulation de tableau ont aussi présenté ce comportement.

Les limites de la traduction syntaxique

On est souvent amené à définir, pour un type donné T_0, plusieurs types synonymes, disons $T_1 \cdots T_n$, chacun représentant un ensemble particulier de données que l'on veut différentier mais que l'on traite concrètement comme des éléments de T. C'est le cas, dans CertiCartes par exemple, pour les différents types d'indices (`bytecode_idx`, `heap_idx`, etc.) qu'on définit comme des synonymes de *nat*.

Or, l'abstraction de ces types peut poser problème. À partir du moment où on veut abstraire T_i vers un type $\hat{T}_i$ et T_j $(i \neq j)$ vers un type $\hat{T}_j$ différent de $\hat{T}_i$ nous introduisons les difficultés suivantes :

- le JTK ne respecte pas la $\delta-$conversion : les types abstraits ne sont plus nécessairement $\delta-$convertibles ;
- il n'est pas facile d'attribuer le bon type à tout motif de T_0 : on ne sait donc pas toujours décider quelle fonction d'abstraction appliquer. L'inférence de type implémentée donnant la $\delta-$forme normale du type considéré, c'est la fonction associée à ce type (si elle est définie) qui sera choisie.

Nous avons rencontré ces cas de figure dans le cadre du chapitre 6. Les solutions que nous avons implémentées sont les suivantes :

- récrire la spécification de façon à ce que certains types cessent d'être des synonymes
- ou corriger, par des commandes d'abstraction, certaines omissions du processus d'abstraction.

Sur un autre plan, il serait intéressant d'étudier et d'implémenter des algorithmes d'inférence de type plus fins permettant une gestion plus appropriée des synonyme de types.

5.2.3 Le processus de nettoyage

La phase de nettoyage est conduite en deux temps. Premièrement, les deux vérificateurs de contraintes (de bonne formation et de typage) parcourent la spécification abstraite `S`. Toute expression JSL violant les contraintes propres du langage est supprimée. Ces suppressions sont répercutées de la façon suivante :

- soit t une expression JSL, la suppression d'un sous-terme de t entraîne la suppression de t ;
- soit c la condition $l \twoheadrightarrow r$, la suppression de l ou de r entraîne la suppression de c ;
- soit r la règle $l_1 \twoheadrightarrow r_1, \ldots, l_n \twoheadrightarrow r_n \Rightarrow g \rightarrow d$, la suppression de d entraîne la suppression de la règle r ;
- une fonction f dont la liste de règles se retrouve vide par ce processus de nettoyage devient une fonction variable.

Finalement, chaque fonction $\hat{f}$ est de nouveau analysée, nettoyée des règles qui sont soient dupliquées soient cibles de commande de suppression.

5.2.4 Le processus de transformation

Cette phase du processus d'abstraction cherche à discriminer les fonctions dont le comportement pourrait poser problème lors de la connexion de JaKarTa avec d'autres systèmes. Rappelons par exemple que le système Coq n'accepte que les fonctions totales et déterministes. Coq rejette donc les fonctions partielles et les fonctions non-déterministes. Ce sont ces deux types de comportements, néanmoins acceptés par JaKarTa, que la troisième phase du processus d'abstraction permet de mettre en évidence. Les commandes de transformation, qui sont prises en compte à cet endroit, permettent de transformer une fonction donnée de façon à ce que ces comportements soient gommés.

Nous allons décrire dans le reste de la section ce qu'on entend par test et résolution du non-déterminisme, par test et résolution de la partialité et nous conclurons par la mise en place de ces mécanismes au sein de la phase de transformation.

Considérations préliminaires Nous ne considérerons dans cette section que les fonctions JSL qui respectent un format particulier garantissant que cellesci possèdent bien une structure arborescente. Ce sont ces contraintes qui, par exemple, permettent aux fonctions JSL d'être soumises avec succès à la compilation vers les langages possédant des opérateurs de filtrage de motifs arborescents comme OCaml ou Coq (voir section 5.3.1).

Un exemple simple de fonctions JSL que nous ne considérons pas ici sont celles qui contiennent par exemple deux règles dont les premières conditions n'ont pas le même membre gauche, comme :

```
function f : T1 → T2 → T :=
```

```
(g x) → True       ⇒ f x y → (h y);
(k y) → False      ⇒ f x y → (h′ x).
```

Par contre nous considérerons la fonction f sous la forme suivante

```
function f : T1 → T2 → T :=
(g x) → True                  ⇒ f x y → (h y);
(g x) → z , (k y) → False     ⇒ f x y → (h′ x).
```

Formellement, nous disposons d'un prédicat *arborescent* qui vérifie s'il existe un parcours arborescent des listes de règles définissant une fonction JSL. Seules les fonctions passant avec succès ce test sont considérées.

Test de non-déterminisme

Ce test a pour objectif de vérifier pour chaque paire de règle (R_1, R_2), contenue dans la définition d'une fonction f, s'il n'existe pas une substitution qui exhiberait un chevauchement de ces deux règles. De façon tout à fait classique l'existence d'un recouvrement de deux règles s'exprime en terme de solution à un problème d'unification. Étant données deux règles

$$R_1 \equiv l_1^1 \twoheadrightarrow r_1^1,\ \dots\ , l_n^1 \twoheadrightarrow r_n^1 \Rightarrow g \rightarrow d^1$$

$$R_2 \equiv l_1^2 \twoheadrightarrow r_1^2,\ \dots\ , l_m^2 \twoheadrightarrow r_m^2 \Rightarrow g \rightarrow d^2$$

avec $m \leq n$ celles-ci se recouvrent lorsqu'il existe un unificateur identifiant $l_1^1 \twoheadrightarrow r_1^1,\ \dots\ , l_m^1 \twoheadrightarrow r_m^1$ avec $l_1^2 \twoheadrightarrow r_1^2,\ \dots\ , l_m^2 \twoheadrightarrow r_m^2$. Dans ce cas le choix entre ces deux règles n'est plus déterministe.

Prise en charge du non-déterminisme

La correction apportée à ce comportement est partielle : seule une sous-classe des fonctions non-déterministes est cible de la transformation en fonctions déterministes. C'est celle des fonctions non-déterministes dont les paires de règles (R_1, R_2) se chevauchant ont même corps. Ces cas sont exactement ceux que nous avons rencontrés dans les différentes abstractions que nous avons définies (se référer au chapitre 6). C'est pourquoi nous nous sommes restreint à cette configuration. Là encore, les mécanismes implémentés sont classiques : l'ensemble des conclusions des règles se recouvrant sont regroupés dans une liste.

Soit $f : \tau_1 \rightarrow\ \cdots\ \rightarrow\ \tau_m\ \rightarrow \tau$ la fonction dont on veut corriger le comportement non-déterministe. La transformation consiste à partitionner l'ensemble des règles définissant f selon la propriété P : "*a même corps*″. Chaque partition définit une nouvelle règle.

Soit p la partition

$$\{corps \Rightarrow g \rightarrow d_1,\ \cdots,\ corps \Rightarrow g \rightarrow d_k\}$$

la règle induite par p est alors $corps \Rightarrow g \rightarrow [d_1 \cdots d_k]$. Celle-ci est donc la règle qui collecte dans une liste tous les résultats possibles pour une même séquence de conditions.

La signature de la fonction f doit alors être adaptée et devient donc $\tau_1 \rightarrow \cdots \rightarrow \tau_m \rightarrow (list\ \tau)$. Le changement de type est alors propagé aux fonctions appelantes de façon tout à fait classique. Cette transformation est à rapprocher de l'application de la monade *list*. Ajoutons ainsi que cette propagation nous a obligé à considérer la construction de fonctions de type *map* afin de prendre en compte le passage aux listes.

Test de partialité

Il s'agit ici de tester si une fonction f est partielle ou totale. Le procédé est relativement intuitif. Il s'agit :

- de reconstruire, à l'image du prédicat *arborescent*, la structure arborescente associée à la liste des règles. Cela correspond à passer d'une représentation en liste de règles vers une définition par filtrage de motifs. Signalons que cette traduction n'est pas toujours possible ;
- puis de vérifier si chaque nœud de l'arbre considère tous les motifs possibles. Dans le cas négatif, la fonction est partielle.

Prise en charge de la partialité

La modification du comportement d'une fonction partielle $f : \tau_1 \rightarrow \cdots \rightarrow \tau_m \rightarrow \tau$ suit les mêmes lignes que celles du non-déterminisme :

- modifier la fonction f de façon à ce qu'elle soit totale, ceci entraîne la modification de son type vers $\tau_1 \rightarrow \cdots \rightarrow \tau_m \rightarrow (err\ \tau)$, où *err* est le type dénotant la monade "lift" (dont la définition JaKarTa est `data err = Val 'a | Err`) ;
- puis propager la modification aux fonctions appelantes.

Décrivons un peu plus précisément le processus de modification. Soit P la partition de règles de f suivante :

$$\begin{array}{c} l_1 \twoheadrightarrow r_1,\ \ldots\ ,l_i \twoheadrightarrow r_i^1,\ l_{i+1}^1 \twoheadrightarrow r_{i+1}^1\ \ldots\ ,l_{n1}^1 \twoheadrightarrow r_{n1}^1 \Rightarrow g \rightarrow d^1 \\ \vdots \\ l_1 \twoheadrightarrow r_1,\ \ldots\ ,l_i \twoheadrightarrow r_i^p,\ l_{i+1}^p \twoheadrightarrow r_{i+1}^p\ \ldots\ ,l_{np}^p \twoheadrightarrow r_{np}^p \Rightarrow g \rightarrow d^p \end{array}$$

Remarquons que ces règles ont un préfixe commun. Elles caractérisent un sous-arbre de la représentation par filtrage de motif. Supposons que l'ensemble $\{r_i^1, \ldots, r_i^p\}$ ne couvre pas tous les cas possibles. Le principe de la transformation consiste alors à rajouter la règle suivante

$$l_1 \twoheadrightarrow r_1,\ \ldots\ ,l_i \twoheadrightarrow x \Rightarrow g \rightarrow Err$$

où x est une variable fraîche jouant le rôle du wildcard "_" de OCaml, puis de modifier la conclusion t de toute règle de P en $Value\ t$.

Le processus est récursif et traite donc une à une toutes les manifestations de partialité.

Mise en place

Soit $\hat{\mathtt{S}}$ la spécification abstraite regroupant donc toutes les fonctions abstraites après la phase de nettoyage. Chaque fonction de $\hat{\mathtt{S}}$ passe le test de partialité, puis le test de non-déterminisme. Si elles sont requises par l'utilisateur, les corrections de comportement sont ensuite effectuées.

5.2.5 Les commandes d'abstraction

Voici la syntaxe des commandes d'abstraction :

$$<\!\textit{commande}\!> \; ::= \; <\!\textit{action}\!> \; <\!\textit{cible}\!>\!\textit{list} \; [\textbf{by} \; <\!\textit{expr}\!>] \; [\textbf{in} \; <\!\textit{portée}\!>\!\textit{list}]$$

où :
- Le champ optionnel *portée* détermine dans quelle partie de la spécification JSL l'action doit être réalisée ; par défaut la portée est la spécification dans son entier.
- Le champ *cible* spécifie quelle composante syntaxique est visée par l'action.
- Le champ optionnel *expr* est utilisé dans les commandes de substitution pour spécifier par quelle expression les cibles doivent être remplacées.

La forme des champs *portée* et *cible* est définie de la façon suivante, où $<\!f\!>$ représente un nom de fonction :

$$\begin{array}{lll}
<\!\textit{cible}\!> & ::= & <\!f\!> \mid <\!f\!>.<\!\textit{loc}\!> \mid <\!f\!>\texttt{@}<\!\textit{nat}\!> \mid <\!\textit{expr}\!> \\
<\!\textit{portée}\!> & ::= & <\!f\!> \mid <\!f\!>.<\!\textit{loc}\!> \\
<\!\textit{loc}\!> & ::= & <\!\textit{nat}\!> \mid <\!\textit{nat}\!>.<\!\textit{nat}\!>.<\!\textit{nat}\!>
\end{array}$$

Une illustration du sens de *portée* et de *cible* est donnée par les exemples suivants :
- `f.2` indique la seconde règle de réécriture de la définition de la fonction `f` ;
- `f.2.3` indique la troisième condition de `f.2`. Par convention, `f.2.0` pointe vers la conclusion de `f.2` ;
- `f.2.3.1` indique le membre gauche de `f.2.3` et `f.2.3.2` le membre droit ;
- `f@i` pointe vers le $\mathtt{i}^e$ argument de `f`.

Les valeurs du champ cible indiquent sur quels éléments d'une spécification la commande agit.

Pour certains éléments, il est de plus possible d'indiquer si l'action a lieu au début ou à la fin de la phase de traduction syntaxique. Au début, la commande agit sur la version concrète de la fonction considérée, à la fin du processus de traduction l'action a lieu sur la version abstraite. Si la différence peut paraître subtile, elle est, dans certains cas, essentielle. En effet, les violations de contraintes introduites par les deux *modi operandi* sont potentiellement différentes.

Les valeurs de portée introduites par le mot-clé `in` permettent de limiter l'action d'une commande à une fonction (*fname*), une règle (*ruleloc*) ou un atome (*absloc*).

Nous introduisons dans les figures 5.8, 5.9 et 5.10 les différentes commandes d'abstraction où :

- *fname*, *argloc*, *absloc*, *exp*, *ruleloc* et *typexp* désignent respectivement les portée et cible suivantes : une fonction `f`, un argument d'une fonction désigné par sa position `f@i`, une expression désignée par sa localisation dans la spécification `f.i.j.k`, une expression, une règle désignée par son numéro `f.i` et une expression de type. Finalement *lfname*, *largloc*, *lexp* désignent respectivement une liste de symboles de fonction, une liste d'arguments d'une fonction particulière et une liste d'expressions.
- Un identificateur $<f>$ précédé de `&` représente la fonction abstraite générée par la phase de traduction syntaxique à partir de *f*.
- Un symbole de fonction précédé par "`" doit être considéré comme une expression (et non un identificateur *fname*).

Commandes d'abstraction	**Actions**
`coercion` *lfname1 [*`in` *lfname2]*	*Les fonctions listées dans lfname1 sont utilisées comme fonctions de conversion en cas de problème de typage (coercions)*
`select` *largloc [*`in` *lfname]*	*Pour chaque argloc* `f@i`*, substitue tout appel à* `f` *par le* `i`^{ème} *argument de cet appel.*
`replace` *ltarget* `by` *target [*`in` *loc]* Combinaisons possibles : *fname* — *exp,fname* — $\emptyset$ *absloc* — *exp* — $\emptyset$ *fname* — *fname,exp* — *fname* *exp* — *exp* — *absloc*	*Remplace les éléments de la liste ltarget par target.*
`replace_type` *largloc* `by` *typexp*	*Permet de modifier la signature d'une fonction. Ainsi* `replace_type` *change le type des éléments de largloc en typexp. La conclusion est désigné par* `f@0`*.*

FIG. 5.8 – Commandes de substitution

Finalement nous sommes en mesure de définir précisément ce que nous entendons par script d'abstraction.

Définition 17 (Script d'abstraction)
Un script d'abstraction est la donnée d'une spécification d'abstraction $\mathcal{A}$ et d'un ensemble de commandes d'abstraction $\mathcal{H}$.

Nous verrons dans la section suivante comment sont interprétées les commandes, et en particulier que la gestion des commandes interprétées lors de la première phase repose intégralement sur la notion de position dans une spécification. Les champs *portée*, *cible* et *loc* permettent de spécifier de telles positions.

Dans ce qui suit, nous supposerons qu'il existe une fonction *loc* qui retourne la position de son paramètre. Nous nous permettrons aussi de parler d'ensemble de positions d'une spécification `S` pour signifier l'ensemble des positions de toutes les expressions (types et sous-expressions inclus) présentes dans la spécification.

Commandes d'abstraction	**Actions**
`drop` *largloc*	*Enlève les arguments de fonctions listés dans largloc. Cette commande accepte le préfixe* `&`.
`delete` *ltarget* *[*`in` *loc]* Combinaisons possibles : *fname,absloc,ruleloc* $\emptyset$ *fname* *fname* *fname,exp* *absloc*	*Efface les (occurrences des) éléments référencés dans ltarget.*
`remove` *lexp [*`in` *lfname]*	*Pendant la phase de traduction, toute règle dont la conclusion contient un sous-terme exp* $\in$ *lexp est effacée.*
`reject` *lexp [*`in` *lfname]*	*Pendant la phase de transformation, si deux règles aux corps identiques sont trouvées et que la conclusion de l'une d'elle contient un sous-terme exp* $\in$ *lexp, celle-ci est effacée.*

FIG. 5.9 – Commandes de suppression

Commandes d'abstraction	**Actions**
`determine` *lfname*	*Rend déterministes les fonctions listées dans lfname et répercute ce changement le long du graphe de dépendance des éléments de lfname.*
`totalize` *lfname*	*Rend totales les fonctions listées dans lfname et répercute ce changement le long du graphe de dépendance des éléments de lfname.*

FIG. 5.10 – Commandes de transformation

5.2.6 Interprétation des commandes

Les commandes `select`, `replace`, `replace_type`, `drop`, `delete` et `remove`

L'interprétation de ces commandes a lieu pendant la première phase du processus d'abstraction.

Construction de la fonction de substitution Le JTK traite de manière interne les commandes de suppression **`delete`** et **`remove`** comme des commandes de substitution. En effet, pour une expression donnée, la suppression directe de celle-ci et sa substitution par une expression qui viole le format JSL ont exactement le même effet :

- dans le premier cas, lors de la phase de traduction, l'expression est retirée et l'action est propagée pendant la phase de nettoyage : le JTK efface la condition, la règle ou transforme la fonction en variable, le cas échéant ;
- dans le deuxième cas, lors de la phase de traduction, l'expression est rem-

placée par l'expression illégale. La phase de nettoyage la repère, l'efface et le JTK propage cette suppression.

Concrètement, considérons les données suivantes :

- l'ensemble $\mathcal{L}oc$ des positions de la spécification `S` ;
- l'ensemble $\mathcal{E}_\perp$ des expressions JSL étendu avec le terme $\perp$, où nous admettons que $\perp$ soit une expression illégale, c'est-à-dire systématiquement refusée par la phase de vérification ;
- un script d'abstraction s ;
- une commande c du script s parmi **`select`**, **`replace`**, **`replace_type`**, **`drop`**, **`delete`** et **`remove`** ;
- l'ensemble $l = \{p_1 \dots p_n\}$ ($\subseteq \mathcal{L}oc$) des positions des expressions sur lesquelles c agit

alors l'expression e_i^c à placer à la position p_i est calculé de la façon suivante :

Dans le cas de **`replace`** et **`replace_type`**, l'option "**by** e" détermine trivialement les e_c. En effet $\forall i. e_i^c = e$.

Dans le cas où $c =$**`select`** `f@j`, soit $(f\ t_1 \dots t_m)$ le terme à la position p_i alors $e_i^c = \lceil t_j \rceil$.

De façon similaire, pour la commande $c =$ **`drop`** `f@j`, soit $(f\ t_1 \dots t_m)$ le terme à la position p_i alors $e_i^c = \lceil (f\ t_1 \dots t_{j-1}\ t_{j+1} \dots t_m) \rceil$.

Finalement, dans le cas de **`delete`** et de **`remove`**, $\forall i. e_i^c = \perp$.

Les commandes de substitution et de suppression considérées sont traduites en une fonction $\mathcal{S}ubs : \mathcal{L}oc \to \mathcal{E}_\perp$ des positions dans les expressions JSL augmentée du symbole $\perp$. Formellement $\mathcal{S}ubs$ est définie comme la plus petite fonction de $\mathcal{L}oc$ vers $\mathcal{E}_\perp$ tel que :

$$\forall i \in \{1 \dots n\}.\ (p_i, e_i^c) \in \mathcal{S}ubs$$

Interprétation $\mathcal{S}ubs$ permet ainsi une mise en place uniforme et simple du traitement des commandes lors de la phase de traduction. Pour une expression donnée e de position pos dans `S`, l'abstraction de e par le JTK est soit $\mathcal{S}ubs(pos)$, si $\mathcal{S}ubs$ est défini en pos, soit le résultat du processus de traduction syntaxique $\lceil e \rceil$, dans le cas contraire.

La commande `drop` La seule exception à l'uniformité de ce traitement est le cas de la commande **`drop`** `f@i` : en plus de la construction de la fonction $\mathcal{S}ubs$ le JTK retire le physiquement le i^e paramètre de l'abstraction de la fonction `f`. L'absence du paramètre `&` indique que la suppression du paramètre a lieu avant la phase de traduction syntaxique. Ainsi la fonction qui sera abstraite est la version de la fonction `f` réduite du paramètre `i`. Au contraire, sa présence indique que la suppression du paramètre a lieu après la phase de traduction syntaxique, c'est-à-dire sur la version abstraite de `f`.

La commande `coercion`

La commande **`coercion`** est interprétée durant la phase de nettoyage. Elle permet de modérer l'action de la vérification du typage en proposant des fonctions de coercions pour la résolution locale d'incompatibilités de type. Soit c la commande **`coercion`** `g`, où $g : T \to T'$ est une fonction (de coercion). Soit u une expression de type T. Si la vérification de types indique que le type attendu de u est T' alors l'interprétation de la commande c sur le terme u retourne $(g\ u)$ au lieu de provoquer sa suppression.

La commande `reject`

La commande de suppression **`reject`** est interprétée pendant la phase de transformation, lors de la comparaison des règles deux à deux.

Les commandes `determine` et `totalize`

L'interprétation de la commande **`determine`** `f` (respectivement **`totalize`** `f`), qui a lieu pendant la phase de transformation, provoque la transformation f en fonction déterministe (respectivement totale) selon les principes exposés section 5.2.4.

5.3 JPI : support à l'interaction

L'indépendance de JaKarTa sur les différent systèmes de la famille des méthodes formelles prend toute son importance si on est capable de traduire le format JSL vers le langage de ces systèmes. Le module JPI de JaKarTa est une boîte à outils mettant en œuvre l'interactivité de JaKarTa avec ces systèmes.

Les assistants de preuve comme Coq permettent la définition de fonctions totales, structurellement récursives (via les définitions par point fixe et analyse de cas). Il est possible, sous certaines conditions, de compiler les fonctions JSL vers de telles définitions récursives. En revanche, d'autres systèmes ciblés, comme quelques démonstrateurs automatiques, offrent, de fait, un format de définition de fonction proche de celui de JSL.

Selon le système cible, il y a donc deux types de compilations. Dans le cas des langages proches de JSL, la traduction peut se faire directement à partir de la spécification JSL à traduire. Il s'agit dans la plupart des cas d'une phase de *pretty-printing* conventionnelle. C'est le cas par exemple de la traduction vers le système Spike ([24]).

Dans le cas des langages avec filtrage de motifs, une traduction vers un format intermédiaire est réalisée. Ce format, appelé JIR (pour *JaKarTa Intermediate Representation*), supporte les définitions récursives.

5.3.1 Le format JIR

Le format JIR est une légère extension des expressions JSL comprenant les définitions par cas.

Définition 18 (format JIR)
Soit Σ une signature d'expressions.

- *L'ensemble $\mathcal{E}_c$ des* expressions JIR *(sur Σ) est défini par la syntaxe suivante :*

$$\mathcal{E}_c := \mathcal{V} \mid \mathcal{E}_c == \mathcal{E}_c \mid \mathcal{C}\ \mathcal{E}_c^* \mid \mathcal{D}\ \mathcal{E}_c^* \mid\ \mathsf{case}\ \mathcal{E}_c\ \mathsf{of}\ \{\mathcal{M}\}$$

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

$$\mathcal{M} := \langle\rangle \quad | \quad \mathcal{M}|\mathcal{P} \Rightarrow \mathcal{E}$$

- *Une* règle de réécriture JIR *est une règle de réécriture de la forme*

$$(f\ x^+) \to e$$

où $x^+ = (x_1, \ \ldots \ , x_n)$ sont des variables deux à deux distinctes et $e \in \mathcal{E}_c$ vérifie $\mathsf{var}(e) \subseteq \{x_1, \ \ldots \ , x_n\}$. L'ensemble des règles de réécriture JIR est noté $\mathcal{R}_{\mathrm{JIR}}$.

De JSL vers JIR Le module d'interaction JPI permet la transformation des fonctions déterministes et totales JSL vers des fonctions JIR.

Il s'agit ici de transformer des listes de listes en structure arborescente. Alors qu'à tout arbre on peut trouver une représentation "plate" sous forme de liste de listes (la liste exhaustive des chemins de la racine aux feuilles de l'arbre considéré), cela n'est plus le cas pour la réciproque. La transformation peut donc échouer.

Si la transformation réussit, un ensemble de f-règles de réécriture est traduit vers une définition récursive "à la Coq".

La traduction suit le même modus operandi que le test de partialité. L'idée est de construire un terme JIR en parcourant les règles de façon arborescente.

De JIR vers JSL Pour conclure, l'opération de traduction de JIR vers JSL est aussi implémentée. Elle transforme les définitions par cas en règles de réécriture conditionnelle. Cette fonctionnalité permet par exemple la traduction de spécifications Coq vers JSL. C'est donc une fonctionnalité importante permettant la réutilisation de formalisations écrites en Coq.

L'opérateur de traduction de JIR vers JSL transforme une structure arborescente en liste de listes. Cette fois-ci l'opération est relativement aisée : construire la liste de tous les chemins possibles de la racine aux feuilles.

Traduction des spécifications JaKarTa vers des langages externes Que ce soit du format JSL ou du format JIR, la traduction vers les langages des systèmes ciblés ne pose que des problèmes techniques mineurs et spécifiques aux langages cibles choisis.

Actuellement les traductions vers OCaml, Coq, Isabelle, SPIKE et PVS sont disponibles.

5.4 JAK : support à la preuve

L'un des objectifs qui a guidé la définition du format JSL était de proposer un format qui facilite

1. le raisonnement sur les spécifications écrite dans ce langage,
2. le développement de procédures de preuve (automatique) pour ce propos.

Le module JAK, pour JaKarTa Automation Kit et développé par P. Courtieu (voir [16]), est la boite à outils fournissant ce support au raisonnement. Par opposition aux autres modules, la plupart des outils JAK sont destinés au système de preuves Coq. Cependant les principes sous-jacents sont applicables aux autres systèmes.

Le support aux preuves Coq est une fonctionnalité en phase de développement. Néanmoins, les résultats préliminaires obtenus se sont montrés probants. Pour preuve, le support aux démonstrations par inversion sur les fonctions JSL a permis une automatisation non négligeable.

Pour chaque fonction f, un principe d'inversion est dérivé à partir de l'analyse du corps de la fonction. Étant donné un prédicat ϕ qui relie la valeur du résultat de f à la valeur des paramètres $\vec{x}$ de f, le principe d'inversion permet de casser la preuve de $\forall \vec{x} : \vec{\sigma}.\ \phi\ (\vec{x},\ f\ \vec{x})$ en plusieurs sous-buts plus aisés à prouver.

Nous abordons sommairement dans cette section les principes sous-jacents à cette automatisation. Les références [16, 18] apportent de plus amples détails sur la mise en place et l'utilisation de ces fonctionnalités.

5.4.1 Preuves par induction sur les fonctions définies par filtrage

En Coq, lorsqu'on choisit un style fonctionnel de spécification, on utilise les fonctions récursives au lieu de relations inductives. Dans ce cas les principes d'élimination, qui sont les principes généraux permettant les démonstrations par analyse de cas, ne sont pas automatiquement disponibles. Néanmoins, une fois la fonction définie (c'est-à-dire une fois que la terminaison de la fonction ait été garantie), la construction d'un tel principe est assez naturelle. Celui-ci est défini en prenant compte fidèlement de la structure interne de la fonction en question. Ce principe permet le *raisonnement par cas/induction sur les branches possibles* de la définition de la fonction.

Prenons un exemple. La fonction suivante :

```
Fixpoint isfourtime [n:nat] : bool :=
 Cases n of
  O      ⇒ true |
  (S n1) ⇒
     Cases n1 of
      O      ⇒ false |
      (S n2) ⇒
         Cases n2 of
          O      ⇒ false |
          (S n3) ⇒
             Cases n3 of
              O      ⇒ false |
              (S n4) ⇒ (isfourtime n4)
```

```
            end
        end
    end
end.
```

rend le booléen vrai si l'argument est un multiple de 4, faux sinon. Coq, de fait, permet la définition suivante pour alléger la définition de telles fonctions.

```
Fixpoint isfourtime [n:nat] : bool :=
 Cases n of     O ⇒ true
              | (S S S S m) ⇒ (isfourtime m)
              | _ ⇒ false
 end.
```

Cette fonction fait appel à une analyse de cas imbriquée. Le principe d'induction associé met en jeu une analyse par cas imbriquée de profondeur 5 (sur les valeurs 0 et (Sn)). Prouver une propriété en considérant chaque cas possible est fastidieux et inintéressant. Or il est évident qu'une analyse simple des cas où les valeurs sont 0, 1, 2, 3 et $4 + m$ est suffisante.

Le module JaKarTa permet de mettre en place automatiquement de tels principes d'induction.

La preuve de la propriété $\forall x.P$ en utilisant `isfourtime` comme modèle sera le terme Coq Q suivant :

```
Fixpoint Q [n:nat]: P :=
<[x:nat](P x)> Cases n of
   | O ⇒ (?: (P O))
   | (S O) ⇒ (?: (P (S O)))
   | (S S O) ⇒ (?: (P (S S O)))
   | (S S S O) ⇒ (?: (P (S S S O)))
   | (S S S S m) ⇒ ((?:(x:nat)(P x) → (P (S S S S x))) m (Q m))
 end.
```

où les cinq `(? :H)` représentent les propriétés `H` qui restent à prouver. Remarquons que la branche de `isfourtime` contenant un appel récursif possède maintenant une hypothèse d'induction.

Une fois construit, le terme de preuve incomplet peut être appliqué à un but de preuve donné :

- par la tactique Coq `Refine`. Dans ce cas des obligations de preuves, permettant de compléter le terme, sont générées ;
- en complétant ce terme vers un principe d'induction réutilisable et général.

Afin d'obtenir le principe général cité :

1. nous remplaçons les `?` par des variables (les `Hypx` de l'exemple ci-dessous) ;
2. nous abstrayons le principe `P` de la propriété.

Dans notre exemple, cela donne :

```
[P:(nat → Prop); Hyp0:(P O);  Hyp1:(P (S O));
 Hyp2:(P (S (S O))); Hyp3:(P (S (S (S O))));
 Hyp4:(x:nat) (P x) → (P (S (S (S (S x)))))]
Fixpoint Q [n:nat]: P :=
<[x:nat](P x)> Cases n of
  | O ⇒  Hyp0
  | (S O) ⇒ Hyp1
  | (S S O) ⇒ Hyp2
  | (S S S O) ⇒ Hyp3
  | (S S S S m) ⇒ Hyp4 m (Q m)
 end
 :   (P:(nat → Prop)) (P O) →  (P (S O)) → (P (S (S O)))
```

```
→ (P (S (S (S O)))) → ((x:nat) (P x) → (P (S (S (S (S x))))))
→ (n:nat)(P n)
```

Pour finir, le module JAK propose une tactique permettant la preuve de propriétés de fonctions par l'utilisation de ce principe d'induction.

5.4.2 Validation d'abstractions

Avec le principe d'induction sur les fonctions, les preuves deviennent relativement facile à conduire. Reste, dans le cadre de la démonstration de la commutation des machines virtuelles, une difficile et fastidieuse tâche : définir le bon lemme de commutation pour chaque fonction f qui a été abstraite en $\hat{f}$. Celui-ci a la forme :

$$\forall x^+.\textit{prémisses} \Rightarrow \lceil (f\ x^+) \rceil = (\hat{f} \lceil x^+ \rceil)$$

où *prémisses* est une liste d'hypothèses statuant sur la forme des paramètres. Ces hypothèses permettent de considérer la commutation de f avec $\hat{f}$ sur les cas acceptables de valeurs que les paramètres peuvent prendre. Par exemple, dans le cas de l'abstraction offensive, nous avons à montrer que les machines défensive et offensive commutent seulement pour des programmes ne provoquant pas d'erreurs de type dans la version défensive. Dès lors les prémisses contiennent une condition de la forme $(f\ x^+) \neq TypeError$.

Il est évident que la forme des prémisses dépend fortement de la nature de l'abstraction, et plus concrètement de la façon dont la fonction est abstraite. Le fait que la méthodologie d'abstraction implémentée soit automatique nous permet de générer la bonne version des lemmes de façon tout aussi automatique. Le JTK est en mesure, pendant l'interprétation de commande d'abstraction au sein d'une fonction f, de déduire la forme des prémisses à ajouter au lemme de commutation de f avec $\hat{f}$.

Prenons un exemple. La commande `remove t` retire toute règle ayant `t` pour conclusion. Une prémisse adéquate et nécessaire pour f, si f a effectivement une règle de ce genre, est alors $(f\ x^+) \neq t$.

Nous avons ainsi étendu le JTK de façon à ce que de tels lemmes soient générés pour chaque fonction abstraite.

Preuves automatiques en SPIKE

Dans un registre différent, S. Stratulat a expérimenté la validation croisée des différentes machines virtuelles en utilisant Spike [34], un démonstrateur de théorèmes basé sur l'induction implicite (se reférer à [24]). À l'image des conclusions tirées par les chercheurs de GEMPLUS [41, 102], cette expérience très positive suggère que les démonstrateurs de théorème du premier ordre peuvent être avantageusement exploités dans le cadre de la vérification de plateforme pour cartes à puce.

5.5 Conclusion

Nous avons présenté dans ce chapitre le système JaKarTa. Celui-ci est un environnement dédié à la conception et à la validation de systèmes de types pour langages de bas niveau comme les plates-formes JCVM, JVM et .NET.

L'outil est composé :

du langage JSL. Nous avons montré comment la réécriture conditionnelle constituait un cadre simple, proche des langages à filtrage de motif et suffisamment flexible pour permettre une manipulation des spécifications JSL. Nous avons aussi défini un vérifieur de type pour JSL ;

du module de calcul d'abstraction JTK. Ce module permet, à partir d'une spécification JSL et des données de l'utilisateur spécifiant l'abstraction à appliquer, de calculer automatiquement la spécification abstraite correspondante ;

de l'interface pour l'interaction JPI. Celle-ci permet de placer JaKarTa comme une plate-forme de spécification générique supportant le calcul d'abstraction. En effet, elle implémente toute une série de traducteurs vers les langages de systèmes qui complètent le processus de vérification formelle (principalement des systèmes de preuve et des langages de programmation) ;

du module de preuve JAK. Il était important que le calcul d'abstraction soit accompagné de fonctionnalités de preuves assistant l'utilisateur dans la démonstration de la validité des abstractions calculées. Nous avons décrit la solution implémentée par les outils proposés par JAK.

5.5.1 Évaluation

JTK *versus* interprétation abstraite Deux approches à la construction d'abstraction étaient possibles :

1. proposer un cadre de construction d'abstractions fidèle aux concepts de l'interprétation abstraite [54], par exemple en adaptant et étendant l'interpréteur de P. Cousot [53]. Cette approche a l'avantage de reposer sur des concepts théoriques bien établis et de permettre une caractérisation précise de la correction d'une abstraction ;
2. proposer un cadre de construction d'abstraction qui se libère des contraintes de l'interprétation abstraite classique mais qui propose les outils nécessaires à la démonstration de correction des abstractions ainsi générées.

Le cahier des charges de JaKarTa nous obligeait à proposer des solutions flexibles et pratiques aux problèmes soulevés par la vérification formelle de plates-formes logicielles. Nous avons donc opté pour la deuxième approche ;

Indépendance de la plate-forme JaKarTa. Le module JPI s'est révélé d'une grande utilité. CertiCartes a ainsi put être traduit en JSL, étudié et manipulé par le JTK, puis retraduit vers Coq. La réutilisation du code s'est révélée d'un indéniable confort ;

Pertinence de la discipline de typage. Le vérifieur de types que nous avons implémenté est simple. Il s'est cependant révélé fort utile. Signalons en particulier son utilisation conjointe avec le JTK. La grande majorité des imprécisions, erreurs de spécification d'abstraction sont interceptées par le vérifieur de types. Dans ce cadre là, le vérifieur joue un véritable rôle de debogueur ;

Limites de JaKarTa. Nous avons commenté, section 5.2.2, les difficultés posées à l'inférence de type soulevées par les types synonymes dans le cadre du calcul d'abstraction. Notre expérience, acquise avec les cas d'étude que nous aborderons au chapitre suivant, nous indique qu'il est possible d'outrepasser de telles limitations. Cependant, il semble judicieux d'étudier des solutions — même partielles, au problème d'inférence de type. Ceci permettrait une construction plus flexible des abstractions.

Ajoutons aussi que nous n'avons pas implémenté de moteur de réécriture conditionnelle. Celui-ci aurait cependant permis l'animation, au sein même de JaKarTa, des spécifications JSL. Nous avons, à ce niveau, préféré déléguer la tâche à OCaml et aux systèmes de réécriture existants. Le JPI fournit à ce propos un support adéquat. Le manque d'un tel moteur ne se situe, en fait, pas au niveau de l'exécution de spécification mais plutôt au niveau du JTK. En effet, nous avons montré que la phase de traduction syntaxique était une phase de réduction de la spécification concrète par les fonctions d'abstraction. Afin de valider cette méthodologie, nous avons préféré nous placer dans un cadre simple. C'est-à-dire, nous avons imposé quelques critères syntaxiques aux fonctions d'abstraction. Ceux-ci garantissent au JTK qu'il n'a besoin que du cadre de la réécriture simple pour réduire les spécification JSL. Il est évident que disposer d'un moteur de réécriture conditionnelle nous permettrait d'affranchir les fonctions d'abstraction de ces contraintes.

Soulignons, malgré tout, que JaKarTa est un outil expérimental qui est encore en phase de développement, comme en témoigne l'évolution rapide du module JAK. Le gain, au niveau de l'automatisation des démonstrations, apporté par ce module ne cesse d'évoluer. Le chapitre suivant montrera que l'outil JaKarTa est cependant fonctionnel, même s'il est amené à beaucoup évoluer. Ce chapitre montrera comment JaKarTa peut apporter son aide aux vérifications formelles comme CertiCartes.

Pour finir, nous exposerons dans le chapitre suivant comment les choix conceptuels et d'implémentation que nous avons été amené à faire ont été validés par les cas d'étude réalisés.

Chapitre 6

Applications de JaKarTa à CertiCartes

Nous allons, dans ce chapitre, présenter l'utilisation de JaKarTa dans le cadre de la vérification formelle de la plateforme JavaCard.

Les chapitres 3 et 4 ont montré à quel point la vérification formelle de plates-formes logicielles de grande taille était laborieuse. Une approche modulaire consistant à dériver de la spécification originale des modèles alternatifs qui se focalisent sur un point particulier (qui coïncide avec la propriété que l'on veut vérifier sur la plate-forme spécifiée) constitue une première rationalisation de l'effort à produire. Cependant ces même chapitres ont souligné le besoin d'outils qui assisteraient le concepteur dans la définition de ces modèles alternatifs et la preuve de correction de ces derniers.

Le chapitre 5 a introduit notre contribution à la résolution de ces problèmes. Le présent chapitre a ainsi pour but de présenter un cas d'étude permettant de mettre à l'épreuve JaKarTa et ainsi de valider la méthodologie sous-jacente.

Organisation du chapitre Nous abordons dans ce chapitre l'application de JaKarTa à la spécification, synthèse par abstraction et validation de machines virtuelles. Concrètement, nous allons présenter la spécification JSL de la machine défensive à la section 6.1. Puis nous montrons comment JaKarTa peut être utilisé pour automatiser le processus de définition des machines virtuelles offensive (section 6.2) et abstraite (section 6.3). Nous montrons aussi comment les fonctionnalités de preuve implémentées par le module JaKarTa permettent d'alléger le processus de validation croisée des machines virtuelles (section 6.4).

Considérations préliminaires

Grâce au module JPI de JaKarTa, nous avons été en mesure d'exporter la formalisation de la dJCVM de Coq vers JSL. Le résultat est sans surprise, nous retrouvons ici la formalisation défensive présentée au chapitre 4. De plus, les machines offensive et abstraite que nous obtiendrons l'ont été de façon à ce qu'elles soient identiques à la version exposée au chapitre 4. Nous n'insisterons donc pas sur les aspects sémantiques des spécifications présentées.

6.1 La machine virtuelle défensive

Puisque nous sommes déjà familiarisé avec la machine virtuelle défensive, nous allons dans cette section décrire sommairement le résultat de la traduction afin de nous familiariser avec la syntaxe JSL.

6.1.1 La modélisation des données de la dJCVM

Commençons par décrire la structure de donnée de la machine virtuelle défensive. Nous retrouvons naturellement la définition des valeurs dynamiques.

```
data valu_prim  =
   VReturnAddress bytecode_idx   |
   VBoolean z                    |
   VByte z                       |
   VShort z                      |
   VInt z .


data valu  =
   VPrim valu_prim  |
   VRef valu_ref .

data valu_ref  =
   VRef_null                                |
   VRef_array type0 heap_idx                |
   VRef_instance cap_class_idx heap_idx     |
   VRef_interface cap_interf_idx heap_idx .

       (* ... *)

data dstate  =
   Build_dstate static_heap heap stack.
```

Remarquons que le type des états de la machine défensive, codé en Coq par un type enregistrement, a été ici traduit en un type inductif à un constructeur. C'est, en effet, sous cette forme que sont gérés, de manière interne, les enregistrements par Coq. La déclaration de ce type est accompagnée par la définition des fonctions accesseurs. Par exemple le champ `sk`, le troisième champ de `dstate`, est accédé par la fonction `stack_f`. Celle-ci est aussi générée automatiquement par Coq et traduite, tout aussi automatiquement, par le JPI.

```
function  stack_f  : dstate → stack :=
 arg0_stack_f → (Build_dstate static_heap_f0 heap_f0 stack_f0)
                        ⇒ (stack_f arg0_stack_f) → stack_f0 .
```

6.1.2 Un exemple d'instruction en syntaxe JSL

Une des conséquences de l'origine Coq de la spécification que l'on manipule est le format particulier du résultat. De fait, toutes les fonctions :
- sont totales ;
- sont déterministes ;
- ont une structure arborescente (au sens de la section 5.3.1).

Toutes ces bonnes propriétés ont, de fait, constitué un cadre confortable à l'utilisation du JTK.

Présentons l'instruction `invokevirtual`. Nous retrouvons ici la sémantique opérationnelle présentée section 4.4.4.

Les règles 1, 2, 3, 6, 7, 8 et 9 traitent des cas d'erreurs : pile vide, incompatibilité de type (la valeur extraite n'est pas une référence), problèmes d'extraction de méthode, de classe, d'objet du tas, et des paramètres de la pile d'opérande.

La règle 4 signale la présence d'un pointeur nul et lance l'exception correspondante. La règle 5 présente la configuration pertinente. C'est le cas où on est en mesure d'effectuer l'appel de la méthode, via la fonction `new_frame_invokevirtual`.

```
function  invokevirtual
    : nat → class_method_idx → dstate → jcprogram → rdstate :=

 <invokevirtual_rule_1>
    (stack_f state) → Nil
          ⇒ (invokevirtual nargs nm state cap)
                             → (abortCode State_error state);

 <invokevirtual_rule_2>
    (stack_f state) → (Cons h lf),
    nargs → O
          ⇒ (invokevirtual nargs nm state cap)
                            → (abortCode Args_error state);

 <invokevirtual_rule_3>
    (stack_f state) → (Cons h lf),
    nargs → (S n),
    (nth_func (opstack h) nargs) → (Some x),
    x → (VPrim v)
          ⇒ (invokevirtual nargs nm state cap)
                             → (abortCode Type_error state);

 <invokevirtual_rule_4>
    (stack_f state) → (Cons h lf),
    nargs → (S n),
    (nth_func (opstack h) nargs) → (Some x),
    x → (VRef vr),
    (res_null vr) → True
          ⇒ (invokevirtual nargs nm state cap)
                             → (throwException NullPointer state cap);

 <invokevirtual_rule_5>
    (stack_f state) → (Cons h lf),
    nargs → (S n),
    (nth_func (opstack h) nargs) → (Some x),
    x → (VRef vr),
    (res_null vr) → False,
    (nth_func (heap_f state) (vr2hi vr)) → (Some nhp),
    (nth_elt (classes cap) (get_obj_class_idx nhp)) → (Some c),
    (get_method c nm) → (Some m)
          ⇒ (invokevirtual nargs nm state cap)
                             → (new_frame_invokevirtual nargs m nhp state cap);

 <invokevirtual_rule_6>
```

```
   (stack_f state) → (Cons h lf),
   nargs → (S n),
   (nth_func (opstack h) nargs) → (Some x),
   x → (VRef vr),
   (res_null vr) → False,
   (nth_func (heap_f state) (vr2hi vr)) → (Some nhp),
   (nth_elt (classes cap) (get_obj_class_idx nhp)) → (Some c),
   (get_method c nm) → None
        ⇒ (invokevirtual nargs nm state cap)
                         → (abortCap Methods_membership_error state);

<invokevirtual_rule_7>
   (stack_f state) → (Cons h lf),
   nargs → (S n),
   (nth_func (opstack h) nargs) → (Some x),
   x → (VRef vr),
   (res_null vr) → False,
   (nth_func (heap_f state) (vr2hi vr)) → (Some nhp),
   (nth_elt (classes cap) (get_obj_class_idx nhp)) → None
        ⇒ (invokevirtual nargs nm state cap)
                         → (abortCap Class_membership_error state);

<invokevirtual_rule_8>
   (stack_f state) → (Cons h lf),
   nargs → (S n),
   (nth_func (opstack h) nargs) → (Some x),
   x → (VRef vr),
   (res_null vr) → False,
   (nth_func (heap_f state) (vr2hi vr)) → None
        ⇒ (invokevirtual nargs nm state cap)
                         → (abortMemory Heap_error state);

<invokevirtual_rule_9>
   (stack_f state) → (Cons h lf),
   nargs → (S n),
   (nth_func (opstack h) nargs) → None
        ⇒ (invokevirtual nargs nm state cap)
                         → (abortCode Opstack_error state).
```

6.2 Synthèse de la machine virtuelle offensive

Nous allons décrire dans cette section comment nous avons utilisé le JTK pour générer la machine virtuelle offensive. Au chapitre 5 nous avons indiqué à partir de quelles données le JTK était en mesure de calculer l'abstraction désirée : la spécification à abstraire, un ensemble de types abstraits, un ensemble de fonctions d'abstraction et un script d'abstraction. Le résultat est une spécification contenant toutes les fonctions qui ont été abstraites.

Nous avons présenté, dans la section précédente, la spécification à abstraire. Nous allons dans les sections suivantes introduire les types abstraits, les fonctions d'abstraction et le script d'abstraction. Notons que ces informations sont très proches de celles que nous avons présentées au chapitre 4, à la différence du script d'abstraction. Nous insisterons donc plus particulièrement sur ce dernier point.

6.2.1 Modélisation des données de la oJCVM

L'abstraction offensive ne considère que la partie numérique de toute valeur manipulée. Ainsi, et sans surprise, nous définissons l'abstraction des types `valu_prim`, `valu` et `valu_ref` comme les types suivant :

```
type ovalu_prim  = z.

type ovalu  =  z.

type ovalu_ref  = z.
```

Nous retrouvons bien ici le modèle offensif présenté section 4.7. L'abstraction des types restants est réalisée de façon semblable.

6.2.2 Fonctions d'abstraction pour la oJCVM

```
function alpha_do_valu_prim : valu_prim → ovalu_prim :=
  ⇒ alpha_do_valu_prim (VReturnAddress v) → (inject_nat v) ;
  ⇒ alpha_do_valu_prim (VBoolean v)       → v ;
  ⇒ alpha_do_valu_prim (VByte v)          → v ;
  ⇒ alpha_do_valu_prim (VShort v)         → v ;
  ⇒ alpha_do_valu_prim (VInt v)           → v .

function alpha_do_valu_ref :  valu_ref → ovalu_ref :=
  ⇒ alpha_do_valu_ref VRef_null              → ZERO ;
  ⇒ alpha_do_valu_ref (VRef_array t hp)      → (alpha_do_heap_idx hp) ;
  ⇒ alpha_do_valu_ref (VRef_instance c hp)   → (alpha_do_heap_idx hp) ;
  ⇒ alpha_do_valu_ref (VRef_interface i hp) → (alpha_do_heap_idx hp) .

function alpha_do_valu : valu → ovalu  :=
  ⇒ alpha_do_valu (VPrim x) → (alpha_do_valu_prim x) ;
  ⇒ alpha_do_valu (VRef r)  → (alpha_do_valu_ref r).
```

Les fonctions `inject_nat` et `alpha_do_heap_idx` sont des fonctions de traduction des naturels vers les entiers relatifs.

6.2.3 Script d'abstraction pour la oJCVM

La figure 6.1 contient le script utilisé pour générer de la JCVM défensive (constituée environ de 10000 lignes de code JSL) la JCVM offensive (environ 5000 lignes). Le script commence avec un préambule qui spécifie la fonction à abstraire, le nom de la fonction résultante (ligne 3) et le nom des fonctions d'abstraction à prendre en considération (ligne 4). Le reste du script est divisé en deux blocs : un bloc de commandes de suppression (ligne 8 à ligne 17) et un bloc de commandes de substitution (ligne 23 à ligne 29) ; il n'y a pas eu à utiliser de commande de transformation dans le cadre de cette abstraction.

Le bloc de commandes entre les lignes 8 et 17 utilise les commandes de suppression **`delete`** et **`drop`**.

- La commande **`delete`** est utilisée pour supprimer toutes les occurrences de la cible dans la portée spécifiée. Par exemple à la ligne 8, le script force la suppression des appels aux opérations de vérification de type (ici `type_verification` ...) partout dans la spécification. Par propagation,

```
(* Preambule *)

build oexec from dexec
abstraction functions alpha_do_valu_prim alpha_do_valu ...

(* Commandes de suppression *)

delete type_verification signature_verification Type_error

       types_match test_exception_aastore assignment_compatible

delete test_exception_putfield_ref.2 test_exception_astore.3

       test_exception_checkcast.4 test_exception_checkcast.5
       test_exception_putstatic_ref.2

delete t0 in putstatic putfield_obj

drop getfield_obj@3 getstatic@1 tableswitch@1 lookupswitch@1

(* Commandes de substitution *)

coercion inject_nat z2n

replace get_inst_from_heap by oget_inst_from_heap
replace v2t by (Ref Ref_null)
replace (trhi2vr (vr2tr vr) (absolu (v2z v)))
                       by v in putfield_obj.36.0

select vp2z@1 v2z@1 vr2z@1 vr2hi@1 extr_from_opstack@2
select tpz2vp@2 in adr_aload res_opstack getfield_obj  ...
```

Fig. 6.1 – Script d'abstraction pour la machine offensive

ces commandes peuvent conduire à l'effacement de règles. En particulier, les règles dont la conclusion a été effacée sont supprimées. Les fonctions concernées par la commande de la ligne 11 statuent sur la légalité de l'exécution de certaines instructions. Parmi les tests effectués se trouve une vérification de type. Cette commande indique au JTK quelles sont les règles qui sont concernées et qu'il faut supprimer. La dernière commande **delete** (ligne 15) est une commande d'optimisation. C'est-à-dire que le résultat de l'abstraction sans cette commande est satisfaisant. Les deux fonctions incriminées effectuent une analyse par cas sur la variable de type `t0`. L'abstraction calculée par le JTK prend en compte correctement cette situation : tous les cas conduisant à une erreur sont supprimés. Cependant il n'est plus nécessaire, pour les différents cas corrects, de garder l'analyse de cas. C'est cette optimisation qu'effectue la commande **delete** ;
- La commande **drop** est utilisée pour effacer certains arguments de la signature d'une fonction. Une telle commande est utile pour nettoyer le code rendu inutile par le processus d'abstraction. Par exemple, la fonction `getfield_obj` prend comme troisième argument un type. Cette information est devenue inutile dans la JCVM offensive. Dès lors, la commande ligne 17 instruit le JTK de la suppression de cet argument.

Le bloc entre les lignes 21 et 29 utilise les commandes de substitution **coercion**, **replace** et **select**.

- **coercion** indique au JTK quelles sont les fonctions de coercion à utiliser en cas de problème de typage. Ces problèmes surviennent naturellement lorsqu'un changement de représentation des données est effectué. La machine défensive gère les adresses de retour (de sous-routines) comme des naturels alors que la machine offensive ne manipule que des valeurs dans $\mathbb{Z}$. Ainsi, dans le cas de la machine offensive, des ajustements sont nécessaires là où les adresses de retour sont encore transmises comme des naturels. La commande de la ligne 21 permet donc d'indiquer quelles sont les coercions autorisées.
- **replace** permet de remplacer la cible par l'argument spécifié dans le champ **by**. Cet argument peut être un nom fonction ou une expression. Cette commande permet à l'utilisateur de passer outre l'expression calculée par le mécanisme standard d'abstraction. Étant donnée la fonction `oget_inst_from_heap`, qui prend en argument un tas `h` et une valeur `v`, et extrait de `h` l'instance référencée par `v`. Nous utilisons la commande de la ligne 23 pour substituer la solution du JTK à l'abstraction de `get_inst_from_heap` par des appels à `oget_inst_from_heap`. Il aurait été possible, mais moins pratique dans ce cas précis, d'utiliser les commandes de suppression afin de produire le même résultat. Observons que seulement trois de ces commandes de substitution sont utilisées pour obtenir une spécification correcte de la JCVM offensive.
- **select** permet l'optimisation de la spécification par la projection d'un appel de fonction sur un de ses arguments. De telles projections sont utiles lorsque l'abstraction produit une fonction `f` qui se comporte comme une fonction de projection, c'est-à-dire telle que $\texttt{f}(x_1, \ldots, x_n) = x_i$ pour tout $x_1 \ldots x_n$. Un exemple d'une telle fonction est la fonction `vp2z`. Dans sa

version défensive cette fonction rend l'entier correspondant à la valeur typée donnée en paramètre. Dans la machine offensive, cette valeur et le paramètre se confondent. `vp2z` est donc l'identité dans la machine offensive, c'est pourquoi, à la ligne 28, nous indiquons que l'abstraction de `(vp2z n)` doit être `n`. Notons que de telles commandes n'affectent que le comportement calculatoire d'une fonction, et non pas son sens.

6.2.4 Résultat de la synthèse de la oJCVM

Observons le résultat de la synthèse sur l'instruction `invokevirtual`. Tout fonction préfixée par `o` est issue de l'abstraction offensive. Ainsi `oinvokevirtual` implémente la sémantique offensive de `invokevirtual`.

Remarquons les points suivant :

- la règle `invokevirtual_rule_3` a disparu. Celle-ci implémentait en effet une vérification de type qui échouait (sur `Type_error`), et a donc été supprimée par l'action de la commande ligne 8 ;
- les occurrences de la condition `x →(VRef vr)` ont disparu. Leur abstraction donne `x →vr`, ce qui est une condition ne servant qu'au renommage de `x`. Le JTK, pendant la phase de transformation sait reconnaître ce type de condition et effectue le nettoyage adéquat (supprimer cette condition et remplacer `vr` par `x` dans le reste de la règle) ;
- la fonction `get_method : class →class_method_idx →(exc method)` reste inchangée. En effet, aucun de ces arguments n'est sujet à l'abstraction.

```
function  oinvokevirtual
      : nat → class_method_idx → ostate → jcprogram → rostate :=

 <oinvokevirtual_rule_1>
     (ostack_f state) → Nil
          ⇒ (oinvokevirtual nargs nm state cap)
                             → (oabortCode State_error state);

 <oinvokevirtual_rule_2>
     (ostack_f state) → (Cons h lf),
     nargs → O
          ⇒ (oinvokevirtual nargs nm state cap)
                             → (oabortCode Args_error state);

 <oinvokevirtual_rule_4>
     (ostack_f state) → (Cons h lf),
     nargs → (S n),
     (nth_func (oopstack h) nargs) → (Some x),
     (ores_null x) → True
          ⇒ (oinvokevirtual nargs nm state cap)
                             → (othrowException NullPointer state cap);

 <oinvokevirtual_rule_5>
     (ostack_f state) → (Cons h lf),
     nargs → (S n),
     (nth_func (oopstack h) nargs) → (Some x),
     (ores_null x) → False,
     (nth_func (oheap_f state) (z2n x)) → (Some nhp),
```

```
    (nth_elt (classes cap) (oget_obj_class_idx nhp)) → (Some c),
    (get_method c nm) → (Some m)
         ⇒ (oinvokevirtual nargs nm state cap)
                    → (onew_frame_invokevirtual nargs m nhp state cap);

<oinvokevirtual_rule_6>
    (ostack_f state) → (Cons h lf),
    nargs → (S n),
    (nth_func (oopstack h) nargs) → (Some x),
    (ores_null x) → False,
    (nth_func (oheap_f state) (z2n x)) → (Some nhp),
    (nth_elt (classes cap) (oget_obj_class_idx nhp)) → (Some c),
    (get_method c nm) → None
         ⇒ (oinvokevirtual nargs nm state cap)
                           → (oabortCap Methods_membership_error state);

<oinvokevirtual_rule_7>
    (ostack_f state) → (Cons h lf),
    nargs → (S n),
    (nth_func (oopstack h) nargs) → (Some x),
    (ores_null x) → False,
    (nth_func (oheap_f state) (z2n x)) → (Some nhp),
    (nth_elt (classes cap) (oget_obj_class_idx nhp)) → None
         ⇒ (oinvokevirtual nargs nm state cap)
                           → (oabortCap Class_membership_error state);

<oinvokevirtual_rule_8>
    (ostack_f state) → (Cons h lf),
    nargs → (S n),
    (nth_func (oopstack h) nargs) → (Some x),
    (ores_null x) → False,
    (nth_func (oheap_f state) (z2n x)) → None
         ⇒ (oinvokevirtual nargs nm state cap)
                           → (oabortMemory Heap_error state);

<oinvokevirtual_rule_9>
    (ostack_f state) → (Cons h lf),
    nargs → (S n),
    (nth_func (oopstack h) nargs) → None
         ⇒ (oinvokevirtual nargs nm state cap)
                           → (oabortCode Opstack_error state).
```

6.3 Synthèse de la machine virtuelle abstraite

L'abstraction vers la aJCVM est plus délicate, et donc plus compliquée à mettre en place. Nous allons dans les sections suivantes illustrer la mise en place de cette abstraction en insistant sur les instructions les plus représentatives : une instruction de branchement (`ifnull`) et une instruction de d'appel de méthode (`invokevirtual`).

6.3.1 Modélisation des données de la aJCVM

Commençons par les types de la machine abstraite. Voici un extrait de la version JSL des types abstraits :

```
data avalu_prim  =
    aReturnAddress bytecode_idx |
    aBoolean    |
    aByte       |
    aShort      |
    aInt .

data  avalu  =
    aPrim avalu_prim       |
    aRef avalu_ref
and
data  avalu_ref  =
    aRef_null  |
    aRef_array avalu       |
    aRef_instance cap_class_idx         |
    aRef_interface cap_interf_idx .

(*  ...  *)

type  astack  =  (list aframe).

data  astate  =  Build_astate  astack.
```

Notons le mot clé **and** qui permet ici l'introduction de définitions mutuellement récursives.

6.3.2 Fonctions d'abstraction pour la aJCVM

Étant donnés les types abstraits, les fonctions d'abstraction sont assez évidentes à définir. En voici des exemples :

```
function alpha_da_valu_prim : valu_prim → avalu_prim :=
  ⇒ alpha_da_valu_prim (VReturnAddress v)  → aReturnAddress v ;
  ⇒ alpha_da_valu_prim (VBoolean v)        → aBoolean ;
  ⇒ alpha_da_valu_prim (VByte v)           → aByte ;
  ⇒ alpha_da_valu_prim (VShort v)          → aShort ;
  ⇒ alpha_da_valu_prim (VInt v)            → aInt .

function alpha_da_valu_ref :  valu_ref → avalu_ref :=
  ⇒ alpha_da_valu_ref VRef_null                  → aRef_null ;
  ⇒ alpha_da_valu_ref (VRef_array t hp)          → aRef_array (t2at t) ;
  ⇒ alpha_da_valu_ref (VRef_instance c hp)       → aRef_instance c ;
  ⇒ alpha_da_valu_ref (VRef_interface i hp)      → aRef_interface i .


function alpha_da_valu : valu → avalu :=
  ⇒ alpha_da_valu (VPrim x)      →  aPrim (alpha_da_valu_prim x) ;
  ⇒ alpha_da_valu (VRef r)       →  aRef (alpha_da_valu_ref r) .

(*...*)

function alpha_da_stack : stack → astack :=
  ⇒ alpha_da__stack  Nil → Nil;
  ⇒ alpha_da__stack (Cons x y) → (Cons (alpha_da__frame x) Nil).

function alpha_da : dstate → astate :=
  ⇒ alpha_da (Build_state sh hp s) → Build_astate (alpha_da_stack s) .
```

Une partie des subtilités propres à la machine virtuelle abstraite se retrouve capturée dans la définition des fonctions d'abstraction. Ainsi :

– la machine virtuelle abstraite garde la trace des adresses de retour nécessaires à la gestion des sous-routines, mais oublie les autres valeurs : la définition de `alpha_da_valu_prim` reflète ce point ;
– selon la fonction `alpha_da_stack`, seule la frame au sommet de la pile d'exécution est conservée ;
– les valeurs abstraites ne comportent aucun composant correspondant au tas des objets et au tas statique, comme l'illustre la définition de `alpha_da`.

L'argument `t` de `VRef_array t hp` est de type `type0` et est abstrait en `avalue` par la fonction auxiliaire `t2at : type0 →avalu`.

Remarquons finalement que nous conservons, pour les états de la machine, un type enregistrement. Or, dans la section 4.8, le type `astate` est défini comme celui des contextes d'exécution abstraits. Disposer d'un type dédié à la représentation des états, par opposition à un type (le type `aframe`) qui représente à la fois les états et les contextes d'exécution, a facilité processus d'abstraction.

6.3.3 Script d'abstraction pour la aJCVM

Nous avons décrit une partie des fonctions d'abstraction utilisées pour générer la machine virtuelle JavaCard abstraite (dont la taille avoisine les 5000 lignes) à partir de la machine défensive.

La figure 6.2 contient une partie du script utilisé pour la dérivation de la machine abstraite JavaCard. Le script dans sa totalité fait un peu plus de 250 lignes. Le préambule est immédiatement suivi par un ensemble de commandes qui agissent sur toute la spécification. Ces commandes sont utilisées pour retirer toute opération et vérification impliquant le tas ou des valeurs. Par exemple `res_null`, `throwException` et `abortMemory` à la ligne 10. De telles commandes sont suffisantes pour gérer correctement un nombre significatif de bytecode, comme `load`, `store`, `new`, `astore`, `goto`, `jsr` et `ret`.

Le reste du script traite, un à un, les instructions dont la sémantique abstraite doit être plus finement réglée. Sont inclues toutes les instructions d'invocation de méthode, de branchement et de gestion de la mémoire.

Résolution du non-déterminisme Prenons comme premier exemple les instructions de branchement. C'est leur sémantique abstraite qui introduit le non déterminisme dans la JCVM abstraite. À la différence de ce qui a eu lieu en Coq, c'est directement l'instruction responsable du comportement incriminé qui règle le problème. En Coq, c'était la fonction `aexec` qui se chargeait de collecter les valeurs possibles (se référer à la section 4.8). Voici comment les commandes d'abstraction permettent de prendre en compte le problème :

– les instructions `ifnull` et `ifnonnull` dépilent une valeur de la pile d'opérande et testent si celle-ci est nulle ou pas. De ce test, la nouvelle valeur du pointeur de programme est dérivée. L'abstraction considérée ne permet plus à ces instructions de statuer sur la valeur dépilée et les instructions résultants doivent prendre en compte simultanément les deux résultats possibles. La commande **`determine`** permet de gérer le non-déterminisme

```
(* Preambule *)

build aexec from dexec
abstraction functions alpha_da_prim alpha_da ...

(* Commandes generales *)

...

delete throwException res_null  abortMemory  ...

...

(***** Commandes pour les bytecodes de branchement *******)

...

determine ifnull ifnonnull
determine res_scompare res_acompare

...

(***** Commandes pour le bytecode invokevirtual *******)

replace (get_obj_class_idx nhp) by
        (get_avalu_ref_class_idx vr) in invokevirtual

reduce &new_frame_invokevirtual at 3

delete violation_security_invokevirtual

replace new_frame_invokevirtual.2.6.2
   by (aNormal
          (Build_astate
              (Cons (Build_aframe
                       (app_return_type l' (snd (signature m)))
                       (a_locvars h)
                       (a_method_loc h)
                       (a_context_ref h)
                       (S (a_p_count h)))
                    Nil)))

(*****************************************************)

...
```

Fig. 6.2 – Script pour l'abstraction typée (extrait)

résultant. Par exemple la commande de la ligne 18 indique au JTK qu'il doit transformer les instructions impliquées en des fonctions déterministes dont le type de retour sera `(list astate)`. La commande, qui implémente la stratégie standard consistant à collecter dans une liste les valeurs de retour d'une fonction non-déterministe, propage aussi la transformation aux fonctions appelantes. Dans notre cas la transformation affecte aussi la fonction `aexec` ;

– les instructions `if_scmp_cond`, `if_acmp_cond` et `if_cond` dépilent deux valeurs de la pile d'opérande, les comparent et le pointeur de programme est déterminé à partir du résultat de cette comparaison. Le calcul de cette comparaison est réalisé par les fonctions auxiliaires `res_scompare` et `res_acompare`, mais, à l'image du point précédent, l'abstraction rend ces fonctions non-déterministes. En effet ces fonctions sont transformées en des fonctions sans arguments qui rendent à la fois $True$ et $False$. De façon similaire la commande **`determine`** est utilisé pour palier ce phénomène (ligne 19). ;

Nous n'avons pas explicitement abordé les instructions `tableswitch` et `lookupswitch`. Car, quoique plus techniques, celle-ci sont conceptuellement traitées de façon tout à fait semblable aux instructions `if`. La valeur dépilée permettait d'extraire, d'une table, le point de programme à prendre en compte. Dans la mesure où cette information est perdue, nous devons prendre en compte tous les points de programme contenu dans la table.

L'appel de méthodes sans les valeurs Pour ce qui est de l'appel de méthode, les ajustements opérés par les commandes d'abstraction sont les suivants :

– la fonction `get_obj_class_idx` recherche la classe de la méthode à charger à partir de l'objet extrait (`nhp`). Dans la version abstraite, les objets n'existent plus. Dans ce cadre, c'est à partir de l'information statique (le type de la référence à l'objet, `vr`) qu'est retrouvée la référence de la classe. La fonction `get_avalu_ref_class_idx` réalise une telle extraction de classe. Le rôle de la ligne 25 est donc d'instruire le JTK de la substitution de `get_obj_class_idx` par `get_avalu_ref_class_idx` ;

Les lignes suivantes du script ont pour objectif de corriger la fonction `new_frame_invokevirtual` qui dans la version défensive se charge d'appeler la méthode invoquée, si cet appel est compatible avec la politique de sécurité du pare-feu, et de préparer la pile d'exécution à cet effet. Dans la version abstraite, rappelons :

– qu'il n'y a que des types et plus de valeurs. Les paramètres et les opérations liés aux valeurs n'ont plus lieu d'être. La ligne 28 retire l'objet des paramètres de la fonction. La ligne 30 supprime les vérifications de sécurité liés à l'objet qui a disparu ;

– que l'exécution est une exécution méthode par méthode (voir section 4.8), c'est à dire qu'un appel de méthode ne résulte pas en son exécution. Dans ce cas seul le type de retour de la méthode appelée est pertinent. La ligne 32 remplace ainsi l'empilement des contexte d'exécution par l'empilement du type de retour dans la pile d'opérande du contexte d'exécution courant.

6.3.4 Résultat de la synthèse de la aJCVM

Voici la version abstraite de l'instruction `invokevirtual` :

```
function  ainvokevirtual
   : nat → class_method_idx → astate → jcprogram → rastate :=

 <ainvokevirtual_rule_1>
     (astack_f state) → Nil
        ⇒ (ainvokevirtual nargs nm state cap)
                          → (aabortCode State_error state);

 <ainvokevirtual_rule_2>
     (astack_f state) → (Cons h lf),
     nargs → O
        ⇒ (ainvokevirtual nargs nm state cap)
                          → (aabortCode Args_error state);

 <ainvokevirtual_rule_3>
     (astack_f state) → (Cons h lf),
     nargs → (S n),
     (nth_func (aopstack h) nargs) → (Some x),
     x → (aPrim v)
        ⇒ (ainvokevirtual nargs nm state cap)
                          → (aabortCode Type_error state);

 <ainvokevirtual_rule_5>
     (astack_f state) → (Cons h lf),
     nargs → (S n),
     (nth_func (aopstack h) nargs) → (Some x),
     x → (aRef vr),
     (nth_elt (classes cap) (get_avalu_ref_class_idx vr)) → (Some c),
     (get_method c nm) → (Some m)
          ⇒ (ainvokevirtual nargs nm state cap)
                            → (anew_frame_invokevirtual nargs m state cap);

 <ainvokevirtual_rule_6>
     (astack_f state) → (Cons h lf),
     nargs → (S n),
     (nth_func (aopstack h) nargs) → (Some x),
     x → (aRef vr),
     (nth_elt (classes cap) (get_avalu_ref_class_idx vr)) → (Some c),
     (get_method c nm) → None
          ⇒ (ainvokevirtual nargs nm state cap)
                            → (aabortCap Methods_membership_error state);

 <ainvokevirtual_rule_7>
     (astack_f state) → (Cons h lf),
     nargs → (S n),
     (nth_func (aopstack h) nargs) → (Some x),
     x → (aRef vr),
     (nth_elt (classes cap) (get_avalu_ref_class_idx vr)) → None
          ⇒ (ainvokevirtual nargs nm state cap)
                            → (aabortCap Class_membership_error state);

 <ainvokevirtual_rule_9>
     (astack_f state) → (Cons h lf),
     nargs → (S n),
```

```
(nth_func (aopstack h) nargs) → None
     ⇒ (ainvokevirtual nargs nm state cap)
                         → (aabortCode Opstack_error state).
```

Remarquons, à cet effet :

– la disparition des règles `invokevirtual_rule_4` et `invokevirtual_rule_8`. Ces règles traitent de cas dépendant des valeurs qui ont disparu ;
– la disparition des occurrences de `res_null`, `heap_f`. Ces fonctions ont, en effet, elles aussi besoin des valeurs ;
– l'appel à `get_avalu_ref_class_idx` en lieu et place de `get_obj_class_idx`.

Pour ce qui est de l'instruction `ifnull`, l'abstraction des valeurs se retrouve particulièrement illustrée par la règle `aifnull_rule_4`. C'est la règle résolvant le problème de non-déterminisme propre à cette instruction de branchement. La règle est la jonction de l'abstraction de deux règles défensives qui ne différaient que par les conditions `(res_null vr) →True` et `(res_null vr) →False` permettant de décider si branchement il y avait. Ces conditions disparaissent sous l'action de l'abstraction et donc, comme nous l'avons cité plus haut, on obtient deux règles au corps identique. La commande **`determine`** a permis de gérer le non-déterminisme résultant.

```
function  aifnull  : bytecode_idx → astate → (list rastate) :=

 <aifnull_rule_1>
    (astack_f state) → Nil
        ⇒ (aifnull branch state)
               → (Cons (aabortCode State_error state) Nil);

 <aifnull_rule_2>
    (astack_f state) → (Cons h lf),
    (head (aopstack h)) → None
        ⇒ (aifnull branch state)
               → (Cons (aabortCode Opstack_error state) Nil);

 <aifnull_rule_3>
    (astack_f state) → (Cons h lf),
    (head (aopstack h)) → (Some v),
     v → (aPrim v0)
        ⇒ (aifnull branch state)
               → (Cons (aabortCode Type_error state) Nil);

 <aifnull_rule_4>
    (astack_f state) → (Cons h lf),
     (head (aopstack h)) → (Some v),
     v → (aRef vr)
        ⇒ (aifnull branch state)
               → (Cons (aupdate_frame
                          (aupdate_pc
                               (S (ap_count h))
                               (aupdate_opstack (tail (aopstack h)) h))
                           state)
                       (Cons (aupdate_frame
                                 (aupdate_pc
                                        branch
                                        (aupdate_opstack (tail (aopstack h))
                                                          h))
```

```
        state)
    Nil)).
```

6.4 Automatisation de la validation croisée des machines virtuelles

Selon les principes exposés sections 4.7.1 et 4.8.1, nous avons amorcé la preuve de validation croisée des différentes machines synthétisées. Il s'agit, de fait, d'un effort qui n'est pas encore arrivé à son terme et qui sera documenté de façon plus détaillée dans [18].

Les expériences de validation en cours ont suivi l'évolution de l'outil JaKarTa et se sont donc déroulées en deux phases :

1. Définition manuelle des lemmes de commutation et de la démonstration de ceux-ci ;
2. Utilisation des fonctionnalités du JTK étendu par la génération de lemmes allié aux tactiques fournies par JAK pour la démonstration de la validation croisée.

Preuves manuelles La démonstration de la commutation des instructions les plus pertinentes, dans le cadre de la validation croisée, a été mise en œuvre (instructions arithmétiques, instructions de branchement, appel de méthode virtuelle, manipulation de champs).

Si cet effort n'a pas été mené, à ce stade, jusqu'à son terme, soulignons cependant que nous avons guidé le JTK de façon à ce que les machines générées soient aussi proche que possible de leur version du chapitre 4. Pour preuve, les deux seules différences notables entre la machine abstraite Coq et celle obtenue par le JTK sont la gestion du non-déterminisme et la représentation des états de la machine virtuelle abstraite. La version CertiCartes résout le non-déterminisme au niveau de la fonction `aexec` alors que la version présentée dans ce chapitre le fait directement dans les instructions de branchement. La différence se situe donc à un niveau d'appel de fonction près. Pour ce qui est de la représentation des états, garder un type enregistrement avec un unique champ pour l'état permettait une traduction syntaxique plus aisée. Confortés par les preuves préliminaires que nous avons menées, nous pensons que l'effort complet de preuve suivra strictement les même lignes que celles tracées dans le cadre de CertiCartes.

D'autre part nous avons écrit un programme OCaml qui exécute, sur les versions OCaml des machines virtuelles, des programmes JavaCard (eux aussi provenant des successives traductions réalisées par le JCVM tools, Coq et JaKarTa) et vérifie la commutation des exécutions. Les expériences menées ont permis d'affirmer notre conviction sans, bien sur, la confirmer.

Utilisation de la génération de lemmes et des tactiques d'induction sur les fonctions Soulignons, là aussi, que cet effort n'a pas été mené à son terme. Cependant, les tactiques d'induction sur les fonctions apportent un réel gain de productivité à l'effort de preuve. En particulier les scripts de preuve sont significativement plus faciles à construire et leur taille est réduite d'un facteur 10 [16]. De plus, 90% des lemmes de commutation de la machine offensive ont été prouvés automatiquement. Nous poursuivons nos expériences afin de prendre en compte la machine abstraite et de mener plus loin cette automatisation. En particulier :

- nous envisageons d'automatiser le raisonnement équationnel en adaptant les connections existantes entre Coq et les systèmes comme Élan [5] ;
- S. Stratulat a expérimenté la validation croisée des différentes machines virtuelles en utilisant Spike [34]. Les résultats sont probants, Spike a été utilisé pour établir les propriétés de commutation de deux tiers de l'ensemble d'instructions à vérifier, mais nous prévoyons optimiser la connexion JaKarTa-Spike afin de pouvoir mener plus loin la validation automatique (se référer aux travaux de S. Stratulat [24]).

6.5 Conclusion

JaKarTa est construit sur une méthodologie robuste pour la certification de vérifieurs de bytecode. L'outil JaKarTa vise à montrer que, pour une propriété de sécurité P donnée, une implémentation efficace (offensive) et une implémentation de référence (défensive) de la machine virtuelle, vérifiant P pendant l'exécution, coïncident sur les programmes qui ont été acceptés par le vérifieur de bytecode vérifiant la propriété P. Un avantage de taille de notre méthodologie est la possibilité de dériver automatiquement :

- les machines offensive et abstraite (sur laquelle est construit le vérifieur de bytecode) ;
- les lemmes de commutation permettant d'établir la validation croisée des machines produites ;
- une partie conséquente de la démonstration de ces lemmes.

L'objectif de ce chapitre était de montrer comment utiliser JaKarTa dans ce but et de valider ainsi l'approche JaKarTa. Nos conclusions, à ce propos, sont les suivantes :

Indépendance de la plate-forme JaKarTa. Pouvoir naviguer entre les différents systèmes (système de preuves, OCaml, JaKarTa) apporte une indéniable flexibilité. D'une part nous avons pu réutiliser la spécification Coq de la dJCVM, d'autre part disposer d'outils aux fonctionnalités hétérogènes permet d'appréhender la tâche de vérification formelle sous différents angles d'attaque. Disposer d'un éventail plus grand de systèmes pouvant communiquer avec JaKarTa (comme les vérifieurs de modèles, par exemple) semble, à ce propos, un objectif intéressant ;

Pertinence de la discipline de typage. Malgré sa simplicité, le vérifieur de types est un élément central de l'architecture de spécification JaKarTa. Dans le cadre de la synthèse d'abstraction, la plupart des erreurs de types avaient pour origine des imprécisions de spécification et des erreurs de spécification d'abstraction. La vérification de type a ainsi permis une véritable construction interactive guidée de scripts d'abstraction.

Coût des abstractions automatisées. Notre expérience de l'outil suggère le constat suivant : pour jouir pleinement des fonctionnalités d'abstractions il faut disposer d'une spécification adaptée. Ce constat est néanmoins sans surprise et raisonnable. Cela aurait été pécher par trop d'optimiste que de penser qu'aucune discipline de spécification ne fût nécessaire ;

Validation de la méthodologie. JaKarTa offre les fonctionnalités nécessaires à la synthèse d'abstractions. Les résultats obtenus représentent, à notre avis, une avancée. En effet, vérifier formellement la plate-forme JavaCard représente un effort important. De plus, les acteurs impliqués dans la vérification formelle ressentent fortement l'absence de tels outils. Les deux cas d'études présentés dans ce chapitre constituent une validation évidente de la méthodologie JaKarTa.

Soulignons à ce propos et à nouveau un point très particulier de la validation que nous avons menée : celle-ci contemplait la synthèse de machines virtuelles que nous avions déjà obtenues au chapitre 4. L'objectif était de tester la faculté qu'a JaKarTa de permettre la construction d'abstractions réalistes de larges spécifications. La construction du moteur d'abstraction a été, en ce sens, interactive. Bien appréhender ce qu'on attend de l'outil nous a permis de le façonner afin d'arriver à cette fin.

Chapitre 7

Conclusion et Perspectives

L'objectif général de cette thèse était de contribuer à l'étude formelle de plates-formes logicielles et en particulier de la plate-forme JavaCard. Pour ce faire, nous devions déterminer, d'une première formalisation de la plate-forme, quelles étaient les besoins, en terme de techniques et d'outils, dans le domaine de la vérification formelle de plates-formes logicielles.

Dans ce cadre, les travaux présentés dans cette thèse se divisent en trois parties :

- Une formalisation de la plate-forme JavaCard : CertiCartes ;
- La construction d'un outil d'aide à la modélisation correcte de plates-formes logicielles : JaKarTa ;
- Validation de l'outil JaKarTa en prenant CertiCartes comme cas d'étude.

Organisation du chapitre Nous commencerons par résumer et évaluer les différentes contributions introduites dans cette thèse (sections 7.1 et 7.2), puis nous discuterons des travaux que nous envisageons d'entreprendre et d'explorer sur chacune de ces contributions (section 7.3).

7.1 CertiCartes

Nous avons présenté au chapitre 4 une formalisation Coq relativement complète de la plate-forme JavaCard qui introduit, d'une part, une étude de faisabilité approfondie d'évaluation Critères Communs de niveau 7 (CC-EAL 7) et, d'autre part, une méthodologie de validation de vérifieurs de bytecode. Celle-ci est constituée de plusieurs machines virtuelles JavaCard : (1) une machine défensive où les valeurs sont accompagnées de leur type et où les vérifications de typage sont effectuées pendant l'exécution ; (2) une machine offensive où les valeurs ne sont pas typées et qui repose la correction de l'exécution sur une phase de vérification de bytecode ; (3) une machine abstraite qui ne manipule que des types et ne vérifiant que l'adéquation du typage. Cette machine sert de base à la construction d'un vérifieur de bytecode.

La formalisation obtenue est *relativement* complète. D'une part elle couvre l'ensemble des instructions JavaCard et prend en compte les aspects importants de la plate-forme comme le pare-feu, les mécanismes de partage d'objets, la gestion des exceptions. D'autre part elle n'est pas suffisamment précise sur le modèle mémoire — essentiellement les objets transients — sur la prise en compte des méthodes natives et sur le contrôle des ressources.

Nos conclusions sur l'effort de formalisation de CertiCartes sont les suivantes :

Validation de la méthodologie CertiCartes L'infrastructure de vérification formelle propre à CertiCartes et schématisée par la figure 7.1

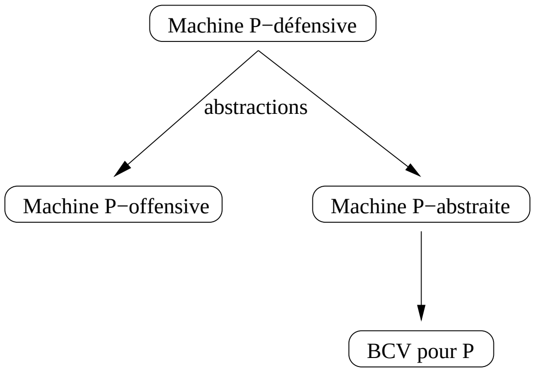

FIG. 7.1 – Méthodologie d'abstraction CertiCartes

a constitué un cadre adéquat et robuste à la vérification de la plate-forme JavaCard. Les avantages de cette méthodologie sont :

1. la machine offensive et la machine abstraite peuvent être dérivées de la machine défensive de façon systématique par des techniques d'abstraction ;
2. la validation, croisée, des différentes machines virtuelles est énoncée schématiquement par : "les machines offensive et défensive coïncident sur les programmes qui passent avec succès la phase de vérification de bytecode" et "les machines abstraite et défensive coïncident sur les programmes qui lancent une erreur de type" ;
3. la correction de la vérification du bytecode est alors une conséquence (a) de la validation croisée des différentes machines virtuelles ; (b) de la correction de sa dérivation à partir de la machine virtuelle abstraite, essentiellement par une analyse de flot de données calculant un point fixe et procédant méthode par méthode.

Soulignons que (a) est spécifique aux machines considérées, mais que (b) est suffisamment générique pour être appliqué à d'autres propriétés, comme par exemple le contrôle de ressources ou du flux d'information (voir [17]). Dès lors la méthodologie est utile pour spécifier et vérifier des langages de bas niveau puisqu'elle permet de distinguer les composantes génériques des composantes spécifiques et puisqu'elle fournit un cadre à la construction de ces dernières ;

Faisabilité de l'évaluation CC-EAL 7 Les laboratoires de recherche de GEMPLUS ont effectué, à ce propos, une évaluation de CertiCartes. Leur constat général est que la méthodologie employée est adéquate, mais que la formalisation n'est pas suffisamment précise. En particulier il est crucial, dans le cadre d'une évaluation EAL 7, de raffiner la formalisation de façon à obtenir une formalisation complète ;

Besoin en terme de support à vérification formelle. Si les assistants de preuve atteignent maintenant un niveau de maturité qui en permet l'utilisation dans un cadre industriel, ceux-ci manquent cependant de fonctionnalités permettant de supporter au mieux les développements de grande taille. Ces développements considèrent l'élaboration d'abstractions afin de factoriser l'effort de vérification. Aussi bien la construction d'abstraction que la démonstration de correction en découlant sont fastidieuses alors qu'il semble qu'une grande partie des lourdeurs soit automatisable et donc évitable.

Ce dernier point nous a conduit à la majeure contribution de cette thèse : la construction du système JaKarTa.

7.2 JaKarTa

JaKarTa, présenté au chapitre 5, est un outil expérimental construit sur une méthodologie robuste pour la conception et la validation de systèmes de types pour langages de bas niveau comme les plates-formes JCVM, JVM et .NET.

Nous avons décrit que l'outil était composé :

du langage JSL. Nous avons montré comment la réécriture conditionnelle constituait un cadre simple, proche des langages à mécanisme de filtrage et suffisamment flexible pour permettre une manipulation des spécifications JSL. Nous avons aussi défini un vérifieur de type pour JSL ;

du module de calcul d'abstraction JTK. Ce module permet, à partir d'une spécification JSL et des données de l'utilisateur spécifiant l'abstraction à appliquer, de calculer automatiquement la spécification abstraite correspondante ;

de l'interface pour l'interaction JPI. Celle-ci permet de placer JaKarTa comme une plate-forme de spécification générique supportant le calcul d'abstraction. En effet, cette interface implémente toute une série de traducteurs vers les langages de systèmes qui complètent le processus de vérification formelle (principalement des systèmes de preuve et des langages de programmation) ;

du module de preuve JAK. Il était important que le calcul d'abstraction soit accompagné de fonctionnalités de preuves assistant l'utilisateur dans la démonstration de la validité des abstractions calculées. Nous avons décrit la solution implémentée par les outils proposés par JAK.

L'outil JaKarTa a été conçu comme un support à l'automatisation de la méthodologie CertiCartes. Ainsi JaKarTa vise à montrer que, pour une propriété de sécurité P donnée, une implémentation efficace (offensive) et une implémentation de référence (défensive) de la machine virtuelle, vérifiant P pendant l'exécution, coïncident sur les programmes qui ont été acceptés par le vérifieur de bytecode vérifiant la propriété P. Un avantage de taille de notre méthodologie, schématisé par la figure 7.2, est la possibilité de dériver automatiquement (1) les machines offensive et abstraite (sur laquelle est construit le vérifieur de bytecode) ; (2) les lemmes de commutation permettant d'établir la validation croisée des machines produites ; (3) d'alléger de façon conséquente la démonstration de ces lemmes.

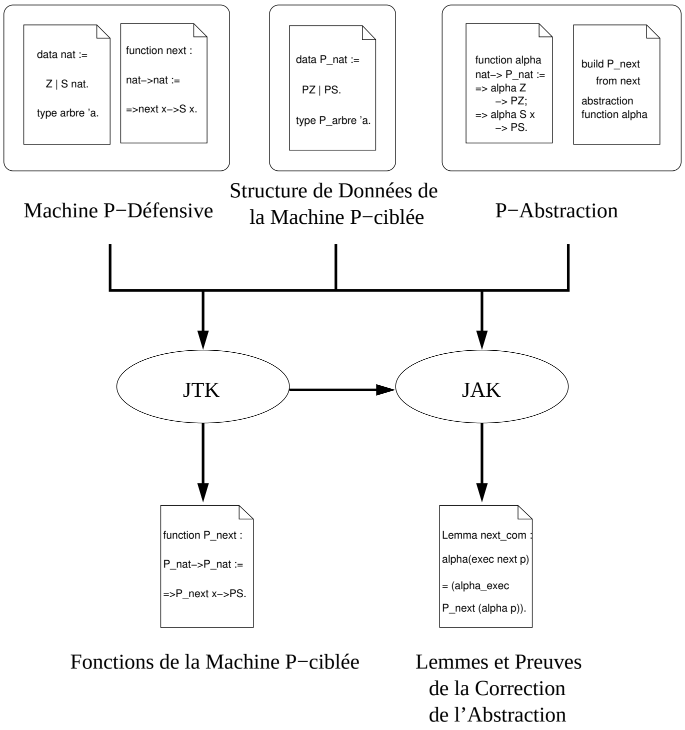

FIG. 7.2 – méthodologie pour l'abstraction via JaKarTa

Évaluation

Nous avons utilisé, au chapitre 6, le système JaKarTa afin d'automatiser les abstractions de la machine virtuelle présentées au chapitre 4.

Nos conclusions, à ce propos, se résument par :

Validation de la méthodologie JaKarTa : Bien que JaKarTa soit un outil encore expérimental et en pleine évolution, les choix de conception et le cahier des charges qui ont conduits son implémentation ont été justifiés.

La neutralité du format JSL alliée à l'interconnexion de JaKarTa avec différents systèmes comme Coq ou OCaml a constitué un cadre privilégié d'utilisation. D'une part les développements Coq antérieurs ont pu être réutilisés, d'autre part la vérification formelle JaKarTa "classique" — spécification et abstraction dans JaKarTa puis démonstration dans un assistant de preuve (Coq), a pu être agrémentée par une phase de tests au niveau d'OCaml.

Sur un autre plan, la discipline de typage de JSL combinée au calcul d'abstraction joue un rôle prépondérant dans l'aide à la construction (interactive) de script d'abstraction ;

Coût des abstractions automatisées : Ajoutons que nous avons été en mesure d'automatiser exactement les abstractions que nous voulions obtenir. Cependant ceci n'a été possible que parce que la formalisation de base avait les propriétés requises. C'est le constat que nous avons dressé au chapitre 6 : pour jouir pleinement des fonctionnalités d'abstractions fournies par JaKarTa, il faut disposer d'une spécification adaptée.

7.3 Perspectives

7.3.1 CertiCartes

Vers une modélisation complète Une extension naturelle des travaux présentés au chapitre 4 est la poursuite et la finalisation de l'effort de modélisation de la plate-forme JavaCard.

Pour ce faire, CertiCartes devra :

- inclure un modèle de la mémoire permettant la prise en compte des objets transients ;
- supporter les transactions ;
- permettre l'intégration et l'exécution des méthodes natives.

Vers de nouvelles machines virtuelles, de nouvelles plates-formes La sécurité de la plate-forme JavaCard repose sur plusieurs fonctions de sécurité qui garantissent, souvent de manière indépendante, une propriété du système. Par exemple, JavaCard repose sur deux automates de sécurité distincts pour garantir le respect des règles du pare-feu (et donc la confidentialité), et l'initialisation des objets préalablement à leur utilisation.

Nous envisageons ainsi d'étendre et d'adapter la formalisation CertiCartes dans le cadre de la définition de nouvelles machines virtuelles défensives de façon à intégrer de tels automates au sein de notre formalisation de la plate-forme JavaCard. Plus précisément et en suivant la méthodologie CertiCartes, nous prétendons augmenter la machine virtuelle de référence avec les automates de vérification de la bonne initialisation des objets et de vérification de non-interférence entre applets. Nous étudierons et définirons ensuite les abstractions permettant d'obtenir les machines virtuelles offensives et abstraites correspondantes. Conforté par les expériences menées au chapitre 6, nous utiliserons l'outil JaKarTa pour mener à terme et automatiquement la construction de telles machines.

Une autre direction envisageable est la formalisation d'autres plates-formes d'exécution comme l'environnement d'exécution CLR des applications .NET [52, 112].

7.3.2 JaKarTa

Nous aimerions proposer avec JaKarTa un environnement complet pour l'automatisation de la validation de la plate-forme et des applications JavaCard. L'outil dans sa phase actuelle est un outil expérimental en constante évolution répondant partiellement à ce desiderata. Ainsi, l'évaluation que nous avons mené au chapitre 6 et le savoir-faire acquis sur la validation de la plate-forme JavaCard nous amène a formuler certaines améliorations et extensions.

D'une part, il nous semble important de pérenniser les acquis. Ceci englobe les efforts visant à (1) augmenter l'ergonomie de l'outil (2) améliorer les différentes composantes de l'outil ou alors amoindrir les contraintes répertoriées.

D'autre part, nous proposons d'étendre l'environnement de spécification vers un environnement (1) disposant des mécanismes nécessaires au développement modulaire et incrémental de spécifications formelles complexes, et (2) intégrant les différents aspects formels liés à la validation, notamment le test, la génération de code, et la vérification formelle.

Ergonomie Une meilleure ergonomie de l'outil nous semble être une étape appréciable dans le cycle de vie de JaKarTa. Ce travail de développement est nécessaire pour lui conférer une plus large audience. Les noms qualifiés permettraient, par exemple, d'obtenir des spécifications plus lisibles. Une aide à l'édition, et plus généralement une interface homme-machine, plus développée en permettrait une utilisation plus aisée.

JSL Certaines extensions du langage, en plus des noms qualifiés, sont nécessaires à une meilleure lisibilité des spécifications de grande taille. Plus précisément nous envisageons d'ajouter des mécanismes de sous-typage, de modules et de surcharge d'opérateurs dans les spécifications JSL. Nous aurions ainsi la possibilité de remplacer, à bon escient, l'utilisation que nous faisions des types synonymes par l'utilisation de sous-types.

JTK Indépendamment de l'inclusion des modules, la prise en charge des types synonymes par le JTK nous semble perfectible. Ainsi un point que nous aimerions étudier est l'intégration d'une meilleure résolution des problèmes de $delta-$convertibilité au sein du JTK. Un autre point que nous aimerions développer est une étude plus approfondie des relations avec le cadre classique de l'interprétation abstraite.

Une des fonctionnalités attendues de JaKarTa que nous avons décrite au chapitre 5 est le support au raffinement. Ce point reste à étudier et à inclure aux outils implémentés par JaKarTa. Le raffinement de CertiCartes en vue de la certification EAL7, pourra servir de cadre à la validation des mécanismes implémentés dans ce sens.

Sur un autre plan, nous aimerions aborder le thème de la spécialisation des spécifications. Il s'agit de développer des techniques et outils permettant d'obtenir, à partir d'une machine virtuelle générique qui implémente de nombreuses caractéristiques de JavaCard, une machine virtuelle spécialisée (et optimisée) avec des fonctionnalités réduites. Les utilisations envisagées sont la construction automatique et correcte de machines virtuelles spécialisées (de telles machines peuvent être requises pour des domaines particuliers, et la conduite manuelle du processus de spécialisation conduit souvent à des erreurs ou à des machines qui ne sont pas optimales, c.à.d. qui font des tests superflus), et le support à une approche multi-vue du système (qui pourra être utile pour une évaluation Critères Communs).

JPI L'implémentation de plates-formes logicielles dans un langage de bas niveau (typiquement C) conduit 1) à l'utilisation d'optimisations subtiles dans le code qu'il est fastidieux de chercher à modéliser formellement et 2) à une interdépendance forte du programme avec son environnement, qui n'est exactement connu que lors de l'exécution.

Dans le cadre d'une certification de la JCVM, il est aussi nécessaire d'établir une relation formelle entre la spécification de haut niveau et sa mise en œuvre logicielle. Afin de répondre à cette exigence, nous envisageons de développer un mécanisme de génération de code qui permettra d'engendrer une implémentation C à partir de modèles formels exprimés en JSL. Un tel mécanisme pourra de plus faire le lien entre la vérification formelle et les systèmes de test. Ce point est important car c'est un des moyens les plus efficaces de mettre en relation une spécification avec une implémentation lorsqu'on considère des systèmes de grande taille. En ce sens, nous reprenons un des slogans fort des travaux de D. Syme et de A. Gordon sur la vérification de la plate-forme .NET [78] :

If you specify, you must test.

JAK Ce module a été fondamental à la réduction du coût de la vérification formelle de spécifications JaKarTa. Dans ce cadre, il est important de poursuivre l'effort d'implémentation de ce module.

Par exemple, l'outil SPIKE ne permet pas de formaliser l'analyse de flot de données sur laquelle repose le vérificateur de bytecode, et a fortiori d'en prouver la terminaison. Dans le cadre d'une évaluation Critères Communs, il est essentiel que les preuves soient intégrées dans un environnement unique, en l'occurrence Coq. Il est donc important de développer des outils permettant de construire à partir d'une preuve SPIKE une preuve en Coq.

Finalement, il serait intéressant d'attaquer le problème de la réutilisation des preuves. Il serait souhaitable de construire des outils qui produisent la preuve de correction d'une spécification à partir d'une preuve de correction de ces composantes, et plus généralement des outils qui reflètent, au niveau des preuves formelles, les mécanismes de spécification modulaire que nous proposons de développer pour les spécifications.

Bibliographie

[1] J.-R. Abrial. *The B-Book : Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, et P. H. Schmitt. The Key approach : integrating design and formal verification of Java Card programs. Dans *Proceedings of the Java Card Workshop, co-located with the Java Card Forum, Cannes, France*, 2000.

[3] R. Alur et D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2) :183–235, Avril 1994.

[4] R. Alur et D. Dill. Automata-theoretic verification of real-time systems. Dans *Formal Methods for RealTime Computing*, Trends in Software Series, pages 55–82. John Wiley & Sons Publishers, 1996.

[5] C. Alvarado et Q-H. Nguyen. elan for equational reasoning in coq. Dans J. Despeyroux, éditeur, *Proceedings of LFM'00*, 2000.

[6] R. Anderson. Why cryptosystems fail, from Communications of the ACM, november, 1994. Dans *William Stallings, Practical Cryptography for Data Internetworks.* IEEE Computer Society Press, 1996.

[7] R. Anderson et M. Kuhn. Tamper resistance – a cautionary note. Dans *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*, pages 1–11, Oakland, California, Novembre 1996. USENIX.

[8] R. Anderson et M. Kuhn. Low cost attacks on tamper resistant devices. Dans *Proc. 5th International Security Protocols Conference*, pages 125–136, 1997.

[9] F. Baader et T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[10] A. Balaa et Y. Bertot. Fix-point equations for well-founded recursion in type theory. Dans M. Aagaard et J. Harrison, éditeurs, *Proceedings of TPHOLS'00*, volume 1689 de *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2000.

[11] BALI project. `http://www4.informatik.tu-muenchen.de/~isabelle/bali`

[12] Bandera project. `http://www.cis.ksu.edu/santos/bandera.`

[13] A. Banerjee et D. A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. Dans *Proceedings of CSFW'02*. IEEE Computer Society Press, 2002.

[14] H.P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*, volume 103 de *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[15] H.P. Barendregt. Lambda calculi with types. Dans S. Abramsky, D. Gabbay, et T. Maibaum, éditeurs, *Handbook of Logic in Computer Science*, volume 2, pages 117–310. Oxford University Press, New York, 1992.

[16] G. Barthe et P. Courtieu. Efficient Reasoning about Executable Specifications in Coq. Dans C. Muñoz et S. Tahar, éditeurs, *Proceedings of TPHOLs'02*, volume 2410 de *Lecture Notes in Computer Science*, pages 31–46. Springer-Verlag, 2002.

[17] G. Barthe, P. Courtieu, G. Dufay, et S. Melo de Sousa. A formal executable semantics of java card : Virtual machines and bytecode verifier. Manuscrit.

[18] G. Barthe, P. Courtieu, G. Dufay, et S. Melo de Sousa. Tool-assisted specification and verification of the javacard platform with jakarta. Manuscrit.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] G. Barthe, D. Gurov, et M. Huisman. Compositional specification and verification of control flow based security properties of multi-application programs. Dans S. Drossopoulou et al, éditeur, *Proceedings of FTfJP'01*, 2001.

[24] G. Barthe et S. Stratulat. Validation of the javacard platform with implicit induction techniques. Manuscrit.

[25] B. Beckert. A dynamic logic for java card. Dans S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, et A. Poetzsch-Heffter, éditeurs, *Proceedings of Formal Techniques for Java Programs.* Rapport Technique 269, 5/2000, Fernuniversität Hagen, 2000.

[26] J. van den Berg et B. 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.

[27] J. Bergstra et J.W. Klop. Conditional rewrite rules : confluence and termination. *Journal of Computer and System Sciences*, 32 :323–362, 1986.

[28] Y. Bertot. Formalizing in Coq a type system for object initialization in the Java bytecode language. Dans G. Berry, H. Comon, et A. Finkel, éditeurs, *Proceedings of CAV'01*, volume 2102 de *Lecture Notes in Computer Science*, pages 14–24. Springer-Verlag, 2001.

[29] Y. Bertot et P. Casteran. Coq'art. `http://www-sop.inira.fr/lemme/Yves.Bertot/coqart.html`

[30] M. Bezem, J. W. Klop, et R. de Vrijer, éditeurs. *Term Rewriting Systems.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2002.

[31] P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, et J.-L. Lanet. Electronic Purse Applet Certification. Dans S. Schneider et P. Ryan, éditeurs, *Proceedings of the Workshop on Secure Architectures and Information Flow*, volume 32 de *Electronic Notes in Theoretical Computer Science.* Elsevier, 2000.

[32] 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.

[33] P. Borras, D. Clement, Th. Despeyrouz, J. Incerpi, G. Kahn, B. Lang, et V. Pascual. CENTAUR : The system. Dans *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (PSDE)*, volume 24, pages 14–24, New York, NY, 1989. ACM Press.

[34] 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.

[35] J. P. Bowen et V. Stavridou. Safety-critical systems, formal methods and standards. *IEE/BCS Software Engineering Journal*, 8(4) :189–209, Juillet 1993.

[36] P. Brisset. A Case Study In Java Software Verification : SecurityManager.checkConnect(). Dans S. Drossopoulou, S. Eisenbach, B. Jacobs,

G. T. LEAVENS, P. MÜLLER, et A. POETZSCH-HEFFTER, éditeurs, *Proceedings of Formal Techniques for Java Programs*. Rapport Technique 269, 5/2000, Fernuniversität Hagen, 2000.

[37] R. M. BURSTALL et J. A. GOGUEN. An informal introduction to specification using CLEAR. Dans Robert S. BOYER et J. Strother MOORE, éditeurs, *The Correctness Problem in Computer Science*, pages 185–213. Academic Press, Inc., New York, NY, 1981.

[38] Camille NG. `http://www.lifl.fr/~grimaud/CamilleNG/CamilleNG.html`

[39] L. CASSET. *Construction Correcte de Logiciels pour Carte à Puce*. Thèse de doctorat, Université de Marseille, 2000.

[40] L. CASSET, L. BURDY, et A. REQUET. Formal Development of an Embedded Verifier for JavaCard ByteCode. Dans *Proceedings of DSN'02*. IEEE Computer Society, 2002.

[41] L. CASSET et J.-L. 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.

[42] 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 22-24 2002.

[43] Z. CHEN. *Java Card technology for Smart Cards : architecture and programmer's guide*. Java series. Addison-Wesley, Reading, MA, USA, 2000.

[44] Y. CHEON et G. T. LEAVENS. A Simple and Practical Approach to Unit Testing : The JML and JUnit Way. Dans B. MAGNUSSON, éditeur, *Proceedings of ECOOP'02*, volume 2374 de *Lecture Notes in Computer Science*, pages 231–255. Springer-Verlag, 2002.

[45] E. M. CLARKE, O. GRUMBERG, et D. A. PELED. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[46] E. M. CLARKE et J. M. WING. Formal methods : State of the art and future directions. *ACM Computing Surveys*, 28(4) :626–643, Décembre 1996.

[47] R. COCKETT et T. FUKUSHIMA. About Charity. Rapport technique 92/480/18, University of Calgary, Juin 1992.

[48] A. COGLIO, A. GOLDBERG, et Z. QIAN. Towards a Provably-Correct Implementation of the JVM Bytecode Verifier. Dans *Formal Underpinnings of Java Workshop at OOPSLA*, 1998.

[49] R. M. COHEN. Defensive Java Virtual Machine Specification Version 0.5. Manuscript, 1997.

[50] COQ DEVELOPMENT TEAM. *The Coq Proof Assistant Reference Manual. Version 7.4*, Février 2003.

[51] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, et H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. Dans *Proceedings of ICSE'00*, pages 439–448. ACM Press, 2000.

[52] Microsoft Corporation. Microsoft il assembly programmer's guide reference manual - part of the .net framework software development, jul 2000.

[53] P. Cousot. The calculational design of a generic abstract interpreter. Dans M. Broy et R. Steinbrüggen, éditeurs, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[54] P. Cousot et R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs. Dans *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM press, 1977.

[55] Common Criteria. `http://www.commoncriteria.org`.

[56] C. J. Date. *The SQL standard*. Addison Wesley, Reading, US, third édition, 1994.

[57] L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, et T. Melham. The PROSPER toolkit. Dans S. Graf et M. Schwartbach, éditeurs, *Tools and Algorithms for Constructing Systems, TACAS'00*, volume 1785 de *Lecture Notes in Computer Science*, pages 78–92. Springer-Verlag, 2000.

[58] S. Drossopoulou et S. Eisenbach. Describing the semantics of Java and Proving Type Soundness. Dans J. Alves-Foss, éditeur, *Formal Syntax and Semantics of Java*, volume 1523 de *Lecture Notes in Computer Science*, pages 41–82. Springer-Verlag, 1999.

[59] G. Dufay. Certification d'abstractions pour la sécurisation de programmes java. Mémoire de D.E.A., Université de Paris 7, 2000.

[60] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, et H. Zheng. Tool-supported program abstraction for finite-state verification. Dans *Proceedings of ICSE'01*, 2001.

[61] R. K. Dybvig. *The Scheme programming language : ANSI Scheme*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, seconde édition, 1996.

[62] e-Europe SmartCards. `http://eeurope-smartcards.org`.

[63] H. Ehrig, W. Fey, et H. Hansen. ACT ONE : An algebraic specification language with two levels of semantics. Rapport technique 83–03, Technical University of Berlin, Fachbereich Informatik, 1983.

[64] C. Flanagan et K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. Dans *International Symposium on FME 2001 : Formal Methods for Increasing Software Productivity*, volume 2021 de *Lecture Notes in Computer Science*, pages 500–517. Springer-Verlag, 2001.

[65] Formal methods ressources. `http://www.afm.sbu.ac.uk/`

[66] S. N. Freund et J. C. Mitchell. A type system for object initialization in the Java bytecode language. Dans *Proceedings of OOPSLA '98*, volume 33(10) de *ACM SIGPLAN Notices*, pages 310–328. ACM Press, Octobre 1998.

[67] S. N. Freund et J. C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6) :1196–1250, Novembre 1999.

[68] K. Futatsugi, J. Goguen, J.-P. Jouannaud, et J. Meseguer. Principles of OBJ-2. Dans B. Reid, éditeur, *Proceedings 12th ACM Symp. on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[69] S. J. Garland, J. V. Guttag, et J. Horning. *An Overview of Larch*, volume 693 de *Lecture Notes in Computer Science*, pages 329–348. Springer Verlag, 1993.

[70] C. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, et J. S. Pedersen. *The Raise Development Method*. Prentice Hall, London, 1995.

[71] A. K. Ghosh. Security risks of Java Cards. Dans *CardTech/SecurTech '98, Volume I : Technology*, pages 465–470, Washington D.C., Avril 1998.

[72] Gibbs. Trends in computing : Software's chronic crisis. *Scientific American*, September 1994.

[73] E. Giménez. A tutorial on recursive types in coq. Rapport technique RT-0221, INRIA, 1998.

[74] E. Giménez et O. 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`

[75] P. Girard. Which security policy for multiapplication smart cards ? Dans *Proceedings of Usenix workshop on Smart Card Technology (Smartcard'99)*, 1999.

[76] J. Goguen. Hidden algebra for software engineering. Dans *Proceedings Combinatorics, Computation and Logic*, volume 21, pages 35–59, Auckland, New Zealand, Janvier 1999. Springer Verlag.

[77] R. Goré et L. Nguyen. CardKt : Automated Logical Deduction on Java Cards. Dans *Proceedings of the Java Card Workshop, co-located with the Java Card Forum, Cannes, France*, 2000.

[78] A. D. Gordon et D. Syme. Typing a multi-language intermediate code. *ACM SIGPLAN Notices*, 36(3) :248–260, 2001.

[79] M.J.C. Gordon et T.F. Melham, éditeurs. *Introduction to HOL : A theorem proving environment for higher-order logic*. Cambridge University Press, 1993.

[80] G. Grimaud. *CAMILLE : Un système d'exploitation ouvert pour carte à microprocesseur*. Thèse de doctorat, Université de Lille, 2000.

[81] G. Grimaud et S. Jean. Code mobile et carte à puce. *Technique et science informatique*, 2002.

[82] Y. Gurevich. Evolving Algebras 1993 : Lipari Guide. Dans E. Börger, éditeur, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[83] U. Hansmann, M. S. Nicklous, T. Schäck, et F. Seliger. *Smart Card Application Development Using Java*. Springer-Verlag, 2000.

[84] J. Harrison. Formal verification in industry. Intel Industry - Type Summer School 1999.

[85] P. Hartel. LETOS - a lightweight execution tool for operational semantics. *Software Practice and Experience*, 29(5) :1379–1416, Septembre 1999.

[86] P. Hartel et L. Moreau. Formalizing the Safety of Java, the Java Virtual Machine and Java Card. *ACM Computing Surveys*, 33(4) :517–558, Décembre 2001.

[87] J. Hatcliff et M. 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.

[88] K. Havelund. Java PathFinder—A Translator from Java to Promela. Dans D. Dams, R. Gerth, S. Leue, et M. Massink, éditeurs, *Proceedings SPIN'99*, volume 1680 de *Lecture Notes in Computer Science*, page 152. Springer-Verlag, 1999.

[89] J. Heering et P. Klint. Semantics of programming languages : a tool-oriented approach. Rapport technique SEN-R9920, CWI, 1999.

[90] Java Technology. `http://java.sun.com`.

[91] JavaCard Technology. `http://java.sun.com/products/javacard`.

[92] T. Jensen, D. Le Métayer, et T. Thorn. Verification of control flow based security policies. Dans *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 89–103. IEEE Computer Society Press, 1999.

[93] JML Specification Language. `http://www.jmlspecs.org`.

[94] C. B. Jones. *Software Development A Rigorous Approach*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1980.

[95] 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.

[96] M. Kaufmann et J. Strother Moore. ACL2 : An industrial strength version of Nqthm. *COMPASS — Proceedings of the Annual Conference on Computer Assurance*, pages 23–34, 1996. IEEE catalog number 96CH35960.

[97] G. Klein et T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 2002. Soumis à publication.

[98] J.W. Klop. Term-rewriting systems. Dans S. Abramsky, D. Gabbay, et T. Maibaum, éditeurs, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford Science Publications, New York, 1992.

[99] Krakatoa. `http://www.lri.fr/~marche/krakatoa/`

[100] K. G. Krauss. *Petri Nets Applied to the Formal Verification of Parallel and Communicating Processes.* Lehigh University. Dissertation, Bethlehem, PA, 1987.

[101] Gemplus Research Labs. Java Card Common Criteria Certification Using Coq. Rapport Technique, 2001.

[102] J.-L. Lanet et A. Requet. Formal Proof of Smart Card Applets Correctness. Dans J.-J. Quisquater et B. Schneier, éditeurs, *Proceedings of CARDIS'98*, volume 1820 de *Lecture Notes in Computer Science*, pages 85–97. Springer-Verlag, 1998.

[103] C. Laneve. A Type System for JVM Threads. *Theoretical Computer Science*, 290 :741–778, 2002.

[104] K. R. M. Leino. Extended Static Checking : A Ten-Year Perspective. Dans R. Wilhelm, éditeur, *Informatics–10 Years Back, 10 Years Ahead*, volume 2000, pages 157–175, 2001.

[105] K. R. M. Leino, J. B. Saxe, et R. Stata. Checking Java programs via guarded commands. Dans B. Jacobs, G. T. Leavens, P. Müller, et A. Poetzsch-Heffter, éditeurs, *Proceedings of Formal Techniques for Java Programs.* Rapport Technique 251, 1999, Fernuniversität Hagen, 1999.

[106] X. Leroy. Java bytecode verification : an overview. Dans G. Berry, H. Comon, et A. Finkel, éditeurs, *Proceedings of CAV'01*, volume 2102 de *Lecture Notes in Computer Science*, pages 265–285. Springer-Verlag, 2001.

[107] X. 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.

[108] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, et J. Vouillon. *The Objective Caml system, release 3.06*, 2002. `http://caml.inria.fr`

[109] The MathWorks. The pentium papers. `http://www.mathworks.com/company/pentium`

[110] A. Mazzeo, N. Mazzocca, S. Russo, C. Savy, et V. Vittorini. Formal specification of concurrent systems : a structured approach. *The Computer Journal*, 41(3) :145–162, 1998.

[111] G. McGraw et E. W. Felten. *Securing Java : Getting Down to Business with Mobile Code.* Wiley, 2nd edition édition, 1999. ISBN 0-471-31952-X.

[112] E. Meijer et J. Gough. Technical overview of the common language runtime.

[113] J. Meyer, P. Müller, et A. Poetzsch-Heffter. The jive system—implementation description. `http://softech.informatik.uni-kl.de/en/publications/jive.html`, 2000.

[114] J. Meyer et A. Poetzsch-Heffter. An architecture for interactive program provers. Dans S. Graf et M. Schwartzbach, éditeurs, *Proceedings of TACAS'2000*, volume 1785 de *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2000.

[115] R. Milner, M. Tofte, R. Harper, et D. MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[116] M. Montgomery et K. Krishna. Secure Object Sharing in Java Card. Dans *Proceedings of Usenix workshop on Smart Card Technology, (Smartcard'99)*, 1999.

[117] J. Strother Moore. Proving theorems about java and the jvm with acl2. Marktoberdorf Summer School 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*, 2001.

[119] P. D. Mosses. Theory and practice of action semantics. Dans *MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science (Cracow, Poland, Sept. 1996)*, volume 1113 de *Lecture Notes in Computer Science*, pages 37–61. Springer-Verlag, 1996.

[120] P. D. Mosses. A tutorial on action semantics. Rapport technique NS-96-14, Basic Research in Computer Science, 1996.

[121] Multos. `http://www.multos.com/`

[122] G. C. Necula. Proof-carrying code. Dans *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.

[123] .NET Card. `http://www.smartcards.net/infosec/.NET.html`

[124] T. Nipkow. Verified Bytecode Verifiers. Dans F. Honsell et M. Miculan, éditeurs, *Proceedings of FOSSACS'01*, volume 2030 de *Lecture Notes in Computer Science*, pages 347–363. Springer-Verlag, 2001.

[125] T. Nipkow et D. von Oheimb. Java_{light} is type-safe—definitely. Dans *Proceedings of POPL'98*, pages 161–170. ACM Press, 1998.

[126] L. Paulson. *Isabelle : a generic theorem prover*, volume 828 de *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[127] M. Pettersson. RML — A new language and implementation for natural semantics. *Lecture Notes in Computer Science*, 844 :117– ? ?, 1994.

[128] A. Poetzsch-Heffter et P. Müller. A Programming Logic for Sequential Java. Dans D. Swiestra, éditeur, *Proceedings of ESOP'99*, volume 1576 de *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.

[129] J. POSEGGA et H. VOGT. Byte Code Verification for Java Smart Cards Based on Model Checking. Dans J.-J. QUISQUATER, Y. DESWARTE, C. MEADOWS, et D. GOLLMANN, éditeurs, *Proceedings of the ESORICS'98*, volume 1485 de *Lecture Notes in Computer Science*, pages 175–190. Springer-Verlag, 1998.

[130] LOOP PROJECT.

[131] C. 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.

[132] Z. QIAN. A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines. Dans J. ALVES-FOSS, éditeur, *Formal Syntax and Semantics of Java*, volume 1523 de *Lecture Notes in Computer Science*, pages 271–312. Springer-Verlag, 1999.

[133] W. RANKL et W. EFFING. *Smart card handbook*. John Wiley and Sons, Inc., New York, NY, USA, second édition, 2000.

[134] W. REISIG. Petri nets and algebraic specifications. Dans *Theoretical Computer Science*, volume 80, pages 1–34. Elsevier, 1991.

[135] A. REQUET. A B model for ensuring soundness of a large subset of the java card virtual machine. *Science of Computer Programming*, 2003.

[136] J. RUSHBY. Formal methods and their role in the certification of critical systems. Rapport technique SRI-CSL-95-1, Computer Science Laboratory, SRI International, Menlo Park, CA, mar 1995.

[137] D. SANNELLA. A survey of formal software development methods. Rapport technique ECS-LFCS-88-56, University of Edinburgh, Juillet 1988.

[138] F. B. SCHNEIDER. Enforceable security policies. Rapport technique TR99-1759, Cornell University, October 1999.

[139] B. SCHNEIER. *Secrets and Lies*. John Wiley, 2000.

[140] G. SÉGOUAT. Preuve en Coq d'une mise en oeuvre de Java Card. Mémoire de D.E.A., Université de Rennes 1, 1999.

[141] B. SERPETTE et M. SERRANO. Compiling scheme to jvm bytecode : a performance study. Dans *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, pages 259–270. ACM Press, Octobre 2002.

[142] N. SHANKAR, S. OWRE, et J.M. RUSHBY. *The PVS Proof Checker : A Reference Manual*. Computer Science Laboratory, SRI International, Février 1993.

[143] Smart cards for windows. `http://www.microsoft.com/smartcards/`

[144] M. H. SØRENSEN et P. URZYCZYN. Lectures on the Curry-Howard Isomorphism. Rapport technique DIKU 98/14, DIKU, 1998.

[145] J. SPIVEY. An Introduction to Z and Formal Specification. *IEEE Software Engineering Journal*, 4(1) :40–50, 1989.

[146] R. Stärk, J. Schmid, et E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation.* Springer-Verlag, 2001.

[147] R. Stata et M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1) :90–137, Janvier 1999.

[148] L. Sterling et E. Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, Mass., second édition, 1994.

[149] M. Strecker. Formal verification of a Java compiler in Isabelle. Dans *Proc. Conference on Automated Deduction (CADE)*, volume 2392 de *Lecture Notes in Computer Science*, pages 63–77. Springer Verlag, 2002.

[150] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.1 Application Programming Interface (API)*, 1999.

[151] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.1 Runtime Environment (JCRE) Specification*, 1999.

[152] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.1 Virtual Machine Specification*, 1999.

[153] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card Platform Security*, 2001. Technical White Paper.

[154] D. Syme. DECLARE : a prototype declarative proof system for higher order logic. Rapport technique UCAM-CL-TR-416, University of Cambridge, Computer Laboratory, Février 1997.

[155] D. Syme. Proving Java type soundness. Dans J. Alves-Foss, éditeur, *Formal Syntax and Semantics of Java*, volume 1523 de *Lecture Notes in Computer Science*, pages 83–118. Springer-Verlag, 1999.

[156] D. Syme et A. D. Gordon. Automating type soundness proofs via decision procedures and guided reductions. Dans M. Baaz et A. Voronkov, éditeurs, *Proc. of LPAR 2002*, volume 2514, pages 418–434. Springer-Verlag, 2002.

[157] S. Thompson. *Haskell : The Craft of Functional Programming.* Int. Comp. Sci. Pearson Edn, 1999.

[158] R. Traunmüller et K. Lenk, éditeurs. *Electronic Government*, volume LNCS 2456. Springer Verlag, 2002.

[159] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, et J. Visser. The Asf + Sdf meta-environment : A component-based language development environment. Dans R. Wilhelm, éditeur, *Proceedings of CC'01*, volume 2027 de *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[160] M. Y. Vardi et P. Wolper. An automata-theoretic approach to automatic program verification. Dans *Symposium on Logic in Computer Science (LICS'86)*, pages 332–345. IEEE Computer Society Press, 1986.

[161] K. N. Verma et J. Goubault-Larrecq. Reflecting BDDs in Coq. Rapport technique RR3859, INRIA projet Coq, Janvier 2000.

[162] D. von Oheimb. *Analyzing Java in Isabelle/HOL : Formalization, Type Safety and Hoare Logic.* Thèse de doctorat, Technische Universität München, 2001.

[163] M. Weiser. Program slicing. Dans *Fith International Conference on Software Engineering, San Diego, California*, Mars 1981.

[164] W. Wu et M. Saeki. Specifying software architectures based on colored petri nets. *IEICE Trans. on Information Systems*, E83-D(4) :701–712, 2000.

Annexe A

Indice des Notations

Nous proposons dans cet indice un ensemble — non exhaustif, des notations Coq et JaKarTa que nous avons utilisé au long de ce document et que nous trouvons utile de rassembler ici. On pourra trouver dans [50] une référence complète pour les notations Coq.

Notations Coq

`Inductive`	introduit un type inductif.
`Mutual Inductive`	introduit des types mutuellement inductifs.
`Record`	type enregistrement.
`Definition`	définition d'un lambda terme (e.g. fonction).
`Variable`	introduction d'une variable.
`Fixpoint`	définition d'un lambda terme (e.g. fonction) par point fixe.
`Lemma`	introduction d'un énoncé (type) dont la preuve (lambda terme) sera construite interactivement.
`Section` t `... end`	déclaration d'une section de nom t.
`Coercion`	déclaration d'une coercion.
`Cases ... of ... end`	notation pour le filtrage.
`Set`	type des ensemble.
`bool`	ensemble des booléens.
`true`	booléen $Vrai$, premier constructeur de `bool`.
`false`	booléen $Faux$, second constructeur de `bool`.
`nat`	ensembles des entiers naturels.
`O`	l'entier 0, premier constructeur de `nat`.
`S`	la fonction successeur, second constructeur de `nat`.
`list`	ensemble des listes polymorphes .
`nil`	la liste vide, premier constructeur de `list`.

`cons`	la fonction ajoutant un élément à une liste, second constructeur de `list`.
`Option`	type de la monade *lift*.
`Some`	premier constructeur de `Option`, dénote l'existence d'un élément : son argument.
`None`	second constructeur de `Option`.
`Exc`	synonyme de `Option` .
`Value`	synonyme de `Some`.
`Error`	synonyme de `None`.
`A*B`	type produit cartésien de A et de B.
`(a,b)`	la paire constituée de a et de b.
`[x:A]b`	notation pour $\lambda x : A.b$.
`(x:A)B`	notation pour $\Pi x : A.B$.
`A->B`	type des fonctions de A vers B, dénote aussi l'implication.

Notations JaKarTa

JSL	JaKarTa Specification Language.
JTK	JaKarTa Transformation Kit.
JPI	JaKarTa Prover Interface.
JAK	JaKarTa Automation Kit.
`function`	définition de fonction.
`definition`	définition de fonction constante.
`variable`	déclaration de variable.
`data`	déclaration de type inductif.
`type`	type abstrait ou type synonyme.
`record`	type enregistrement.