Preuve de correction d'un traducteur de code Java vers JavaCard dans le système Coq*

Nicolas MAGAUD

31 août 1999

Résumé

Une nouvelle génération de cartes à puce, les *SmartCards*, basée sur le langage *Java* a récemment fait son apparition. On souhaite pouvoir développer des applications destinées à des telles cartes sur des machines de bureau. Le transfert du code de la machine de bureau vers la carte requiert une transformation du bytecode de départ (format JVM) en du bytecode JCVM. Compte tenu des caractéristiques des cartes, cette traduction se doit d'être la plus efficace possible en terme de taille du code généré et de rapidité d'exécution. Un algorithme original de traduction a été présenté dans [Pot99]. Nous présentons une formalisation de la partie de l'algorithme correspondant à l'émission de code sous certaines contraintes de type. Après une rapide description de l'algorithme, nous en modélisons la partie *émission de code* et prouvons que le programme traduit simule bien le programme source.

Mots-clé : méthodes formelles, modélisation, Java, JavaCard, Coq

Préambule

Ce rapport présente les travaux que j'ai effectué durant mon stage de deuxième année de magistère. Ce stage s'est déroulé du premier juin au 31 août 1999 au sein de la société *Trusted Logic*, une start-up récente issue de l'action *VIP* (Validation formelle de protocoles Internet et Web sécurisés) du *GIE Dyade*. Elle est spécialisée dans la recherche, le développement et la commercialisation de technologies autour de la sécurité et des systèmes ouverts dans les domaines des cartes à microprocesseur, des terminaux et des systèmes embarqués. Mon sujet de stage se situait plus précisément dans le domaine de la vérification de composants logiciels pour le développement d'applets.

*Une partie de ces travaux est couverte par le brevet français (n° 99 10696) déposé le 23 août 1999 par la société TRUSTED LOGIC.

Table des matières

1 Introduction

Le langage *Java* et plus particulièrement son sous-ensemble *JavaCard* sont de plus en plus utilisés pour le développement d'applications de petite taille destinées aux systèmes embarqués.
Le langage *JavaCard*, développé par *Sun* est destiné à permettre de programmer une nouvelle génération de carte à puces appelées *SmartCards*. Celles-ci contiennent, comme les ordinateurs *classiques*, un processeur ainsi que trois sortes de mémoires, deux de type persistante (EEPROM et ROM) et une de type volatile (RAM). Ces cartes disposent de plus d'une machine virtuelle propre au langage *JavaCard* - la JCVM - et d'un système d'exploitation installés en ROM.
L'EEPROM peut être utilisée pour le téléchargement et l'ajout d'applets (petites applications écrites en *Java*) à la demande en fonction des besoins de l'utilisateur. Ainsi la carte peut jouer à la fois le rôle de carte bancaire, de carte de fidélité ou autres. Cette cohabitation de différentes applications au sein de la carte nécessite de prendre des précautions quant à l'intégrité et la confidentialité des données des applets les unes par rapport aux autres : c'est le rôle du système d'exploitation. Enfin, l'utilisation d'une machine virtuelle permet de développer des applets portables pouvant être installées sur n'importe quelle carte disposant d'une machine virtuelle *JavaCard* [Sun99].

Afin de simplifier le développement de nouvelles applications destinées à être embarquées sur une carte, on souhaite permettre la réalisation de tels programmes sur des machines de bureau. Dans ce cas, le code *Java* écrit est compilé sous forme de bytecode pour la machine virtuelle *Java* (JVM). Cette dernière utilise des mots de 32 bits alors que son homologue *JavaCard* travaille avec des mots de 16 bits. Certaines machines virtuelles *JavaCard* implémentent les instructions sur 32 bits, mais ce n'est pas le cas en général. Par exemple, la JCVM n'implémente *a priori* qu'une instruction d'addition d'entiers 16 bits, alors que la JVM ne dispose que d'une instruction d'addition sur 32 bits. Un programme développé sur une machine de bureau, et par conséquent compilé en bytecode JVM ne sera donc pas directement exécutable par la JCVM. Or, puisque l'application devra fonctionner sur la machine virtuelle *JavaCard* (JCVM), il faut définir un moyen de transformer un programme *Java* en du bytecode exécutable par la JCVM.
Une solution possible pour résoudre ce problème serait d'écrire un compilateur de *Java* vers la JCVM. Cependant, cela demande un travail relativement complexe et coûteux en temps si l'on ne dispose pas déjà d'un compilateur *Java* vers JVM que l'on peut modifier pour qu'il génère du bytecode JCVM. L'alternative proposée est de développer un traducteur de bytecode JVM vers du bytecode JCVM. Cela permet de laisser le libre choix du compilateur à l'utilisateur ; de plus cela conduit à un outil de taille raisonnable dont il sera plus facile d'établir une preuve formelle de correction et ainsi d'avoir la garantie que cette traduction ne modifie pas le comportement de l'application développée.
Du fait du faible espace de stockage disponible et de la capacité réduite des processeurs des cartes, cette traduction se doit d'être la plus efficace possible en termes de taille du code émis, ainsi qu'en rapidité d'exécution sur la JCVM. François Pottier a proposé dans [Pot99] une méthode de traduction pour ce qui concerne tous les calculs arithmétiques. La traduction effective du bytecode JVM en du bytecode exécutable par la JCVM comprend en plus des aspects arithmétiques, la mise en correspondance des noms de classes côté JVM avec ceux utilisés côté JCVM (tokenisation), ainsi qu'une modification du format de stockage du bytecode, sous forme de *class file* pour la JVM et de *CAP file* pour la JCVM.

On se propose de décrire formellement à l'aide de l'assistant de preuve *Coq* [HKPM99, BBC+99] la sémantique des instructions arithmétiques des machines virtuelles *Java* et *JavaCard* ainsi que certains composants du convertisseur de bytecode JVM vers JCVM ; puis, à partir de cette modélisation, d'établir plusieurs propriétés dont une preuve que le programme JCVM obtenu simule bien le programme JVM de départ.

Dans tout ce qui suit, nous ne considérons que la partie de la traduction concernant les instructions arithmétiques. Nous commençons par décrire rapidement le principe du traducteur (Section 2). Dans un deuxième temps, nous présentons la modélisation de la partie *"Arithmétique"* du traducteur en *Coq* (Section 3). Finalement, nous donnons les propriétés démontrées (Section 4) avant de conclure sur le travail réalisé et de présenter les développements supplémentaires possibles.

2 La traduction du code

Le langage *JavaCard* peut être considéré comme un langage à pile disposant de primitives permettant de transférer un mot de la pile vers une variable locale ou un champ d'objet. En particulier, toutes les instructions arithmétiques ont leur(s) opérande(s) sur le sommet de la pile et y déposent leur résultat. Comme la machine virtuelle *Java* n'utilise que des mots de 32 bits (`int`), et que son homologue *JavaCard* fonctionne avec des mots de 16 bits et ne dispose pas toujours d'instructions de calcul sur 32 bits, traduire un programme exécutable par la JVM en un programme exécutable par la JCVM nécessite de déterminer sous quelle forme les quantités manipulées par la JVM pourront être représentées dans la JCVM ; 16 bits (type *Java* `short`) seront-ils suffisants ou devra-t-on utiliser un mot de 32 bits (type *Java* `int`) ? Une fois qu'un choix de format a été fait pour les données, il faut émettre du bytecode JCVM qui corresponde bien aux tailles de mots (`short` ou `int`) choisies et qui conserve la sémantique du programme.

2.1 Les propriétés requises pour le traducteur

Les trois principaux objectifs que doit vérifier le traducteur sont les suivants :

1. La traduction doit conserver la sémantique du programme source. Les comportements observables sur la machine de développement et sur la carte doivent être les mêmes.
2. Le code émis doit être de la meilleure qualité possible, tant du point de vue de la taille du code, que de la vitesse d'exécution et de la consommation de mémoire vive.
3. On doit éviter, dans la mesure du possible d'utiliser des instructions JCVM manipulant des mots de 32 bits, car certaines machines virtuelles *JavaCard* ne les implémentent pas.

La première propriété est bien sûr la plus importante et sera celle que nous montrerons au cours de notre modélisation en *Coq*. Les deux autres propriétés que l'on exige du traducteur ne seront pas prouvées ; il serait d'ailleurs assez difficile de formuler précisément et objectivement ce qu'est un code de bonne qualité - il s'agit nécessairement d'un compromis entre la taille du code et la vitesse d'exécution - . Cependant, ce sont ces conditions qui expliquent la complexité de mise en oeuvre d'un algorithme de traduction performant, comme celui présenté dans [Pot99].

Un algorithme trivial, consistant à conserver toutes les données en 32 bits et à effectuer tous les calculs avec des instructions JCVM ayant des opérandes sur 32 bits serait correct par rapport à la première condition. Par contre, il pourrait ne pas fonctionner sur certaines cartes n'implémentant pas l'arithmétique 32 bits et présenterait une surconsommation de mémoire vive en utilisant toujours un `int` (32 bits) pour coder une valeur, même si elle est représentable par un `short` (16 bits).

2.2 Principe général

L'algorithme de traduction du bytecode JVM vers du bytecode JCVM se compose de deux étapes ; la première correspond à une analyse statique du bytecode JVM. Pour chaque valeur prise par le compteur ordinal, on décide de la forme que peuvent prendre les éléments de la pile. La seconde consiste en l'émission de code *JavaCard* en se basant sur les résultats de l'analyse statique.

2.3 Systèmes de types et de représentations proposés

Les types arithmétiques de base (`short` et `int`) de *Java* ne conviennent pas pour l'analyse. Le programme source *Java* ne contient en effet que des données codées sur 32 bits, alors que la question que l'on se pose est de savoir si une valeur de type `int` est représentable ou non en un point donné du programme par une valeur de type `short`. Pour mener à bien la phase d'analyse, on va utiliser deux systèmes de types ; dans tout ce qui suit, le premier serait désigné par le vocable "type" et le suivant par "représentation".

2.3.1 Le système de type

Il s'agit de trouver des conditions suffisantes pour qu'un entier représenté sur 32 bits puisse être converti en un entier sur 16 bits sans perte d'informations utiles pour le bon déroulement du programme. On définit trois types `ext`, `int`, et `raw` que l'on utilise pour typer des mots de 32 bits.
Ces types sont définis de la manière suivante :

- Une valeur entière codée sur 32 bits admet le type `ext` lorsqu'il est possible de la représenter par un mot de 16 bits sans perte d'informations. Cela signifie que les mots de type `ext` sont des entiers dont les 16 bits de poids fort sont égaux au bit de signe (bit 15) du mot de poids faible.
- Une valeur entière (32 bits) admet le type `int`, quelle que soit sa forme.
- Une valeur entière (32 bits) admet le type `raw` si l'on interdit toute utilisation des bits 16 à 31 (mot de poids fort) dans la suite du calcul. Une valeur peut prendre le type `raw` si le résultat final du calcul ne dépend pas de la valeur de ses bits de poids fort. C'est le cas lorsque l'on a une addition sur 32 bits suivie de l'instruction Java `i2s` qui "jette" les 16 bits de poids fort de la valeur du sommet de pile.

Intuitivement, on voit que l'on doit distinguer deux cas qui autorisent l'abandon des bits de poids fort. Il est évidemment possible d'abandonner ces bits lorsqu'ils sont *redondants* (`ext`). Cependant, on peut aussi abandonner les bits de poids fort lorsque leurs valeurs n'interviennent pas dans la suite des calculs, on les qualifie alors de *non-pertinents* (`raw`).

On définit deux types supplémentaires, `byte` et Top (noté T). Une valeur entière de type `byte` est une valeur dont seuls les bits 0 à 7 sont significatifs ; tous les autres

forment simplement une extension de signe et sont redondants avec le bit 7. Enfin, comme toutes les valeurs manipulées ne sont pas des entiers, on définit un type Top que l'on attribue à toutes les valeurs numériques et à toutes les valeurs non numériques, i.e. les tableaux et les objets.

2.3.2 La relation de sous-typage

Les types définis dans la section précédente peuvent être reliés entre eux par une relation de sous-typage :

$$\texttt{byte} \leq \texttt{ext} \leq \texttt{int} \leq \texttt{raw} \leq Top$$

Il est clair que si les 24 bits de poids fort d'un mot sont redondants avec le bit 7 (type `byte`), alors les 16 bits de poids fort sont redondants avec le bit 15, et donc cette valeur admet le type `ext`. Une valeur de type `ext` est bien du type `int` puisque toutes les valeurs entières admettent le type `int`. Si une valeur a le type `int`, alors elle a effectivement le type `raw`. Il suffit pour cela de s'interdire d'utiliser les 16 bits de poids fort de cette valeur. Finalement, `raw` est trivialement un sous-type de Top qui est le sommet de la hiérarchie de types. Toutes les données manipulées sont en fait de type Top.
Cette relation de sous-typage est utilisée dans la phase d'analyse du code. A partir des schémas de types possibles pour les instructions *Java*, on cherche à inférer le type de la pile pour chaque valeur du compteur ordinal. L'utilisation du sous-typage permet de remplacer cette inférence de types par un algorithme de résolution de contraintes.

2.3.3 Le système de représentation

Une fois que l'analyse statique a produit un type pour la pile i.e. une liste de types (`ext`,`int` ou `raw`), un pour chaque élément de la pile, il est possible d'émettre du code *JavaCard* en se laissant guider par les types. Le code ainsi généré sera optimal en terme de mémoire vive utilisée puisqu'il utilisera toujours le plus petit mot mémoire permettant de stocker une valeur. Cependant, si l'on code sur 16 bits tous les éléments pour lesquels l'analyse avait autorisé le type `raw` ou le type `ext`, on complique parfois inutilement le code par de nombreux changements de format des données (de 16 à 32 bits, et inversement). L'exemple suivant illustre parfaitement ce problème :

```
short s;
int i1,i2;
i1 = s + i2;
```

La variable `s` est représentée sur 16 bits, elle nécessitera donc une conversion (`s2i`) pour pouvoir effectuer l'addition. Comme `s` est le premier argument de l'addition chargé sur la pile et que `s2i` modifie le sommet de pile, on doit obligatoirement faire un échange entre les deux éléments au sommet de la pile. Si de plus l'instruction n'était pas commutative, il faudrait de nouveau échanger les éléments du sommet de pile. Voici comment le bytecode JVM (à gauche) correspondant au programme *Java* ci-dessus serait traduit en code exécutable par la JCVM (à droite) :

```
iload_0     sload_0     // chargement des opérandes sur la pile
iload_2     iload_3
iadd        swap_x 2,1  // addition vs.
            s2i         // (échange de sommet ; conversion ; addition)
            iadd
```

```
istore_1    istore_1    // stockage du résultat
```

Le problème rencontré est que la variable (16 bits) `s` qui va être utilisée dans un calcul sur 32 bits va être convertie tardivement. Elle ne sera malheureusement plus au sommet de la pile à ce moment. En sachant que cette conversion était nécessaire, on peut simplifier le code en choisissant de représenter sur 2 mots JCVM de 16 bits la valeur de `s` dès qu'elle a été chargée sur la pile. En consommant un peu plus de mémoire vive, on peut améliorer sensiblement les performances en terme de taille du code et de vitesse d'exécution.

On ajoute donc à l'analyse de type précédente, une analyse de représentation plus grossière qui permet d'éviter les trop nombreuses conversions de valeurs de 16 à 32 bits, et inversement. Pour chaque valeur définie dans le programme, on doit choisir une représentation `single` ou `double` à chaque point du programme. Ces représentations sont définies de la façon suivante : Une valeur représentée par `single` sera codée sur 16 bits et une valeur représentée par `double` sera codée sur 32 bits. Ces représentations seront inférées par un système de contraintes imposant qu'une valeur stockée en un point particulier sur 32 bits le sera partout ; ainsi on garantit que l'on ne fera pas de conversions répétées de cette valeur.

2.3.4 Liens entre type et représentation

Les notions de types et de représentations ne sont pas indépendantes l' une de l'autre. Une valeur peut voir sa représentation imposée par l'analyse de type. En effet, si l'algorithme de typage a inféré `int` la valeur devra nécessairement avoir la représentation `double` ; si les 32 bits sont significatifs (c'est ce que nous apprend le type `int`), alors on doit représenter la valeur avec `double`. Par contre, une valeur de type `ext` ou `raw` peut admettre l'une des deux représentations `single` ou `double`.

L'analyse de type sert à déterminer quelles quantités doivent nécessairement être stockées sur 32 bits (type `int`), et l'analyse de représentation permet de propager ces résultats par la contrainte que "si un élément est de type `int` (donc nécessairement de représentation `double`), il doit être codé sur 32 bits en tout point du programme". Sans l'analyse de type, l'algorithme d'inférence des représentations pourrait donner la représentation `single` à tous les éléments. C'est la contrainte "`int` → `double`" qui assure la correction de l'algorithme de traduction.

2.4 La phase d'émission du bytecode JCVM

Nous arrivons maintenant à la partie de la traduction qui va plus particulièrement nous intéresser dans la suite de ce rapport. Il s'agit de voir comment émettre du code JCVM en utilisant les informations fournies par les analyses de type et de représentation précédentes.

La traduction sera menée instruction par instruction sur le code du programme. On va supposer qu'à chaque compteur ordinal JCVM peut correspondre une séquence d'instructions dont seule la dernière peut être un branchement (conditionnel ou non). Cela permet d'avoir facilement l'équivalence des programmes vus comme des transformateurs de compteurs ordinaux. Il suffit de montrer que les tests dans les machines JVM et JCVM sont équivalents et produisent donc les mêmes branchements.

Il nous faut traduire chaque instruction JVM en une séquence d'instructions JCVM sémantiquement équivalente. Les analyses de type et de représentation précédentes nous ont fournies les tailles de mots (16 ou 32 bits) à utiliser à chaque instant du

programme pour représenter les données dans la machine virtuelle *JavaCard*. Il ne reste plus qu'à émettre le code correspondant.

Considérons le cas de l'addition `iadd` 32 bits dans la JVM : si la pile, qui contient les opérandes a comme représentation `single` : `single` : ϕ, avant l'exécution de l'instruction d'addition et `single` : ϕ (où ϕ est la représentation du reste de la pile), on traduit par l'instruction JCVM `sadd`. Plus généralement, les instructions dont les représentations sont de la forme `single`* $\rightarrow$ `single`* sont traduites par leurs homologues travaillant sur 16 bits. Celles de représentations `double`* $\rightarrow$ `double`* sont traduites par les mêmes instructions 32 bits. Il serait toujours possible de traduire ces instructions en des séquences d'instructions 16 bits les simulant. L'addition d'entiers 32 bits pourrait être simulée par des additions sur les mots (16 bits) de poids faible et fort correspondants avec propagation de retenue. Cependant, cela donnerait du code de grande taille et peu performant, ce qui serait contraire aux spécifications de l'algorithme. Les contraintes émises par l'algorithme de traduction portent sur la forme des représentations des instructions, par exemple `iadd` : $\forall \alpha, \alpha\ \alpha \rightarrow \alpha$. Cela nous garantit que les sorties de l'analyse statique en terme de représentations seront bien dans les entrées attendues par la partie de l'algorithme traitant de l'émission de code.

3 Modélisation en *Coq*

Nous commençons par définir tous les types de données (entiers machine, types et représentations). Nous discutons ensuite les différentes techniques de modélisation des fonctions partielles en *Coq*. Dans un troisième temps, nous formalisons les notions de piles pour les machines JVM et JCVM, puis les relations les reliant. Finalement, nous décrivons les sémantiques des instructions *Java* et *JavaCard* ainsi que la fonction de traduction du bytecode *Java* vers le bytecode *JavaCard*.

3.1 Les types arithmétiques *Java* et leurs propriétés

On considère deux ensembles d'entiers : le premier, $\mathbb{I}$, contient tous les entiers relatifs compris entre -2^{31} et $2^{31}-1$, le second, $\mathbb{S}$, contient tous les entiers relatifs compris entre -2^{15} et $2^{15}-1$. On définit sur ces ensembles les opérations arithmétiques usuelles en raisonnant respectivement modulo 2^{32} pour $\mathbb{I}$ et 2^{16} pour $\mathbb{S}$. Il serait donc possible de représenter ces ensembles par $\mathbb{Z}/2^{32}\mathbb{Z}$ et $\mathbb{Z}/2^{16}\mathbb{Z}$; cependant il n'existe pas de théorie sur les anneaux quotients $\mathbb{Z}/2^n\mathbb{Z}$ en *Coq*.
Notre choix est "d'axiomatiser" la représentation des entiers machine dans *Coq*. Ils sont simplement définis sous forme de variables de type *Set*, et on ne définit que les éléments dont nous aurons besoin dans la modélisation, par exemple les constantes "zéro" de chacun des deux ensembles $\mathbb{I}$ et $\mathbb{S}$.

```
Parameter java_int : Set. (* I *)
Parameter java_short : Set. (* S *)
Parameter zero_s : java_short.
Parameter zero_i : java_int.
```

Ce choix permettra de pouvoir remplacer ces paramètres par une implémentation réelle des anneaux quotients si une telle distribution apparaît. Il suffira alors de relier les définitions réelles aux paramètres et de remplacer les axiomes posés par des lemmes que l'on pourra prouver. Les instructions d'addition sur les ensembles $\mathbb{I}$ et $\mathbb{S}$ sont elles aussi définies comme des paramètres sans les expliciter. Les axiomes définis dans la suite

permettront de caractériser certaines de leurs propriétés utiles dans la formalisation et les preuves.

```
Parameter iadd : java_int -> java_int -> java_int.
Parameter sadd : java_short -> java_short -> java_short.
```

Il nous faut maintenant relier entre eux les ensembles $\mathbb{I}$ et $\mathbb{S}$. On définit pour cela deux opérations : la projection `pro`, et l'injection `inj`.

- `pro` est une fonction de $\mathbb{I}$ vers $\mathbb{S}$ et vérifie la propriété : $\forall i \in \mathbb{I},\ \forall s \in \mathbb{S},\ pro(i) = s \leftrightarrow s \equiv i\ [2^{16}]$.
- `inj` est une fonction de $\mathbb{S}$ vers $\mathbb{I}$ qui associe chaque $s \in \mathbb{S}$ à lui-même vu comme un élément de $\mathbb{I}$.

```
Parameter inj : java_short -> java_int.
Parameter pro : java_int -> java_short.
```

Finalement, il nous faut ajouter quelques axiomes qui correspondent aux propriétés arithmétiques que nous aurons à utiliser dans les preuves pour montrer que la sémantique est bien conservée par la traduction. Le premier `pro_inj_is_equal` exprime le fait qu'un élément $s \in \mathbb{S}$ que l'on injecte dans $\mathbb{I}$ (i.e. dont on étend le bit de signe au mot de poids fort ajouté) et que l'on projette vers $\mathbb{S}$ redonne l'élément s de départ. En effet, les 16 bits de poids faible ne sont modifiés par aucune des 2 opérations. Le suivant `iadd_sadd` exprime le fait que "additionner deux quantités 16 bits" revient à "étendre ces quantités à 32 bits, à les additionner puis à projeter le résultat sur $\mathbb{S}$".

```
Axiom pro_inj_is_equal : (s:java_short) s=(pro (inj s)).
Axiom iadd_sadd :
  (s,s0:java_short)(pro (iadd (inj s) (inj s0)))=(sadd s s0).
```

3.2 Représentation des fonctions partielles en *Coq*

3.2.1 Première approche

Le système *Coq* ne permet que la définition de fonctions totales. Lorsque l'on a des fonctions partielles à définir, on doit d'abord commencer par les transformer en des fonctions totales. Par exemple, pour définir une sémantique comme une fonction des piles dans les piles (fonction qui sera alors une fonction partielle), on doit ajouter un constructeur $\perp$ au type pile comme on le ferait dans une description sur papier. Cependant, cette méthode conduit à des spécifications et des preuves compliquées avec de nombreux cas particuliers liés à l'introduction des $\perp$. Lors d'une formalisation, on doit faire intervenir les cas indéfinis représentés par des bottoms $\perp$ différents pour chaque type explicitement partout dans la spécification et dans les preuves alors que sur papier, on n'a recours à ce formalisme que lorsque cela est absolument nécessaire. De plus, la représentation sous forme de fonctions conduit à de nombreux raisonnements par cas qu'il est difficile de résoudre efficacement. Enfin, il est très difficile de déterminer la forme nécessairement prise par les variables présentes dans le but ; à chaque fois, il faut faire une induction sur le type de la variable et prouver par contradiction sur une hypothèse que la forme prise rend le contexte inconsistant. La spécification des sémantiques et autres transformations sur les piles JVM ou JCVM s'avère donc fastidieuse et donne une formalisation très lourde et peu utilisable en pratique.
Après plusieurs essais infructueux de modéliser le problème de cette manière, nous avons décidé de représenter les fonctions partielles sous forme de relations et plus précisément sous forme de *prédicats inductifs* dans *Coq*.

3.2.2 Les prédicats inductifs et l'inversion

Avec des relations, on perd la notion d'argument et de résultat qui jouent alors des rôles équivalents. Il est cependant toujours possible d'établir des lemmes exprimant que certaines relations sont fonctionnelles. Les prédicats inductifs sont définis par un ensemble de constructeurs qui donnent des règles de construction des instances valides de ce prédicat. La relation définie par un prédicat inductif est la plus petite relation fermée par les règles données (i.e. les constructeurs).

```
Inductive even : (list nat) -> Prop :=
  | even_O : (even O)
  | even_SS : (n:nat) (even n) -> (even (S(S n))).
```

`even` est naturellement défini inductivement par le fait que zéro est pair et que si un entier n est pair, alors $S(S(n))$ est pair (où S(n) est le successeur de n au sens de l'arithmétique de Peano).

L'intérêt d'utiliser de tels prédicats dans la modélisation est que l'on dispose de tactiques efficaces pour prouver des théorèmes mettant en jeu ces prédicats, les tactiques d'*Inversion*. Ces tactiques sont utilisables dans les situations suivantes :

- Une des hypothèses du contexte est une instance inconsistante d'un prédicat inductif (i.e. elle ne peut pas être construite à l'aide des constructeurs du prédicat), le but courant peut alors être montré en utilisant le fait que "d'un ensemble inconsistant d'hypothèses on peut déduire n'importe quel théorème".
- Une des hypothèses est une instance d'un prédicat inductif, et que celle-ci a des variables dont on voudrait propager les contraintes au but courant ou à d'autres hypothèses.

Considérons l'exemple suivant. On cherche à montrer qu'aucun nombre naturel n'est strictement inférieur à zéro (`O`). On commence par déplier le `not` et introduire les hypothèses dans le contexte.

```
Welcome to Coq V6.3 (July 1999)

Coq < Lemma no_lt_than_zero : (n:nat)~(lt n O).
1 subgoal

  ============================
   (n:nat)~(lt n O)

no_lt_than_zero < Red; Intros.
1 subgoal

  n : nat
  H : (lt n O)
  ============================
   False
```

Le prédicat `lt` est simplement défini à partir de `le`. A ce niveau, une simple inversion sur H permettrait de résoudre le but. Cependant, pour mieux voir le fonctionnement de la tactique Inversion, on déplie le `lt` pour avoir une hypothèse `H` avec le prédicat `le`, qui lui est défini de façon inductive.

```
no_lt_than_zero < Print lt.
lt = [n,m:nat](le (S n) m)
```

```
      : nat->nat->Prop

no_lt_than_zero < Unfold lt in H.
1 subgoal

  n : nat
  H : (le (S n) O)
  ==========================
   False
```

Le prédicat `le` est défini, pour tout n sous la forme d'un prédicat unaire "être plus petit que n". Ce dernier est défini par deux règles qui expriment les faits suivants :

$$n \leq n \quad \& \quad \forall m \in \texttt{nat},\ n \leq m \rightarrow n \leq S(m)$$

Ceci donne bien une caractérisation de la relation $\leq$. L'hypothèse dont on dispose est de la forme `H : (le (S n) O)`. On voit que cette hypothèse, instance du prédicat inductif `le` est inconsistante puisqu'il est impossible de la déduire des règles de `le`. Le but peut donc être prouvé par simple raisonnement par l'absurde. La tactique *Inversion* réalise tout le travail décrit dans ce paragraphe et se révèle très utile pour éliminer tous les cas pathologiques à examiner dans une preuve de plus grande taille ; ce qui était notre problème principal dans l'approche précédente.

```
no_lt_than_zero < Print le.
Inductive le [n:nat] : nat->Prop :=
      le_n : (le n n) | le_S : (m:nat)(le n m)->(le n (S m))

no_lt_than_zero < Inversion H.
Subtree proved!

no_lt_than_zero < Save.
Red; Intros.
Unfold lt in H.
Inversion H.

no_lt_than_zero is defined

Coq <
```

Les définitions de relations sous forme de types inductifs ainsi que l'utilisation de l'inversion sont présentés plus en détail dans [Gim98] et [BBC+99, pages 141 à 144]. L'exemple présenté ci-dessus est extrait de [Gim98, page 22].

3.3 La relation entre piles JVM et piles JCVM

3.3.1 Représentation des piles en Coq

Une pile de la machine JVM est représentée par une liste d'éléments de type `java_int` (entiers machine 32 bits : $\mathbb{I}$), dont la tête correspond au sommet de pile. Les piles de la machine JCVM, qui peuvent contenir indifféremment des éléments de type `java_short` $\mathbb{S}$ et `java_int` $\mathbb{I}$ seront représentées par une liste d'éléments de type `java_mixed`, aussi noté $\mathbb{S}+\mathbb{I}$:

```
Inductive java_mixed : Set :=
    short_to_mixed : java_short -> java_mixed
  | int_to_mixed   : java_int -> java_mixed.
```

```
Definition JVM_stack  : Set := (list java_int).
Definition JCVM_stack : Set := (list java_mixed).
```

3.3.2 La correspondance des piles

Il faut maintenant définir une correspondance entre les piles de mots de 32 bits (*Java*) et les piles mixtes de mots de 16 ou 32 bits (*JavaCard*). On va commencer par définir cette correspondance au niveau des composantes de la pile, puis on généralisera à la pile entière par une induction sur la structure de listes.

Considérons un élément i de la pile JVM qui se traduit en un élément m de la pile JCVM. La "fonction de traduction" dépend à la fois du type inféré et de la représentation. On la divise donc en deux étapes. La première transforme la valeur en se basant sur le type inféré par l'analyse de types alors que la suivante utilise les représentations.

La traduction dirigée par les types Une valeur i de la pile JVM est représentable soit sur 16 bits ($\mathbb{S}$), soit sur 32 bits ($\mathbb{I}$). En fonction du type inféré, on peut déterminer la taille minimale dans laquelle peut être stockée la valeur dans la pile JCVM.

$$[\![\, ext \,]\!] \;=\; \mathbb{S} \qquad [\![\, int \,]\!] \;=\; \mathbb{I} \qquad [\![\, raw \,]\!] \;=\; \mathbb{S}$$

Pour chaque type α, on relie α à $\mathbb{I}$ par deux relations : In_α ($\alpha \;\rightarrow\; \mathbb{I}$) et Out_α ($\mathbb{I} \;\rightarrow\; \alpha$), ces relations permettent de relier les éléments de la pile JVM aux éléments de la pile JCVM correspondants :

- Si i admet le type `ext`, il est représentable dans $\mathbb{S}$. Les 16 bits de poids fort sont redondants avec le bit 15. On peut donc traduire la valeur i de la JVM vers une valeur m de la JCVM telle que i est la représentation de m étendue à 32 bits. On a donc $Out_{\texttt{ext}} = inj^{-1}$, qui n'est pas totale sur $\mathbb{I}$.
- Si i admet le type `int`, il est nécessairement traduit par i dans la JCVM. On a donc $Out_{\texttt{int}} = id_{\mathbb{I}}$.
- Si i admet le type `raw`, il est représentable dans $\mathbb{S}$. Les 16 bits de poids fort sont inutiles pour la suite du calcul. La traduction de i sera donc m telle que $m = pro\ i$.

Les relations In_α et Out_α où α est un type sont réciproques l'une de l'autre :

$$\forall i \in \mathbb{I}, \forall m \in \mathbb{S} + \mathbb{I},\ Out_\alpha\ i\ m \;\leftrightarrow\; In_\alpha\ m\ i$$

L'ensemble des relations, fonctionnelles ou non [1], utilisées dans cette partie de la traduction des piles dirigée par les types sont présentées ci-dessous. Bien que `pro` soit une fonction, sa relation "réciproque" $\texttt{pro}^{-1}$ n'est pas fonctionnelle.

$$In_{ext} \;=\; inj \qquad In_{int} \;=\; id_{\mathbb{I}} \qquad In_{raw} \;=\; pro^{-1}$$

$$Out_{ext} \;=\; inj^{-1} \qquad Out_{int} \;=\; id_{\mathbb{I}} \qquad Out_{raw} \;=\; pro$$

On ne va définir qu'un seul prédicat inductif pour In et Out : `TDT` (pour type-directed-translation). A partir de celui-ci (correspondant à Out), il est facile de représenter In_α par simple échange des paramètres de la relation.

[1] Il est toujours possible d'établir des lemmes montrant que certaines de ces relations sont fonctionnelles.

```
Inductive TDT : type -> java_int -> java_mixed -> Prop :=
  | TDT_ext : (y:java_short)
      (TDT ext (inj y) (short_to_mixed y))
  | TDT_int : (x:java_int)
      (TDT int x (int_to_mixed x))
  | TDT_raw : (x:java_int)
      (TDT raw x (short_to_mixed (pro x))).
```

La taille (en nombre de bits) donnée à une valeur lors de la traduction vers la pile JCVM par l'analyse de types est la plus petite possible. Nous devons maintenant répercuter les résultats de l'analyse de représentation plus grossière dans la deuxième phase de traduction des piles.

La traduction dirigée par les représentations On définit une relation `RDT` qui transforme un élément de la pile JCVM obtenue à partir de l'analyse de types en un élément de la pile "définitive" JCVM sémantiquement équivalente à la pile de départ JVM.

Lors du calcul de la représentation, des données estampillées de type `raw` ou `ext` peuvent être représentées par des entiers sur 32 bits : on a donc besoin d'une nouvelle table de correspondance entre les types et les représentations.

Dans cette étape, on va réagrandir les "boîtes" dans lesquels on stocke les valeurs. Les éléments typés par `ext` ou `raw` et dont la représentation est `double` vont être de nouveau codés sur 32 bits. Cependant, l'information contenue dans leurs 16 bits de poids fort et perdue lors de la traduction dirigée par les types doit être reconstituée. Un élément e, typé `ext`, dont la représentation est `double`, est facilement reconstructible. Les bits de poids fort ont été abandonnés pour cause de *redondance*, il n'y a donc pas eu de perte d'informations. Il suffit de plonger la valeur e de type `java_short` dans l'ensemble des `java_int` en utilisant la fonction inj. Dans le cas d'un élément m, typé `raw` et de représentation `double`, les bits abandonnés étaient *non-pertinents* ; il suffit d'avoir 16 bits de poids fort aléatoires, qui peuvent être différents des 16 bits de poids fort initiaux. D'après le lien entre représentation et type, le type `int` n'est en relation qu'avec la représentation `double`. Dans tous les autres cas, les éléments ne sont pas modifiés par cette transformation.

	ext	int	raw
$single$	$id_{\mathbf{S}}$	$-$	$id_{\mathbf{S}}$
$double$	inj	$id_{\mathbf{I}}$	pro^{-1}

TAB. 1 – Fonctions de conversion en fonction des types et représentations

```
Inductive RDT : type -> representation -> java_mixed -> java_mixed -> Prop :=
  | RDT_ext_single : (x:java_short)
      (RDT ext single (short_to_mixed x) (short_to_mixed x))
  | RDT_ext_double : (x:java_short)
      (RDT ext double (short_to_mixed x) (int_to_mixed (inj x)))
  | RDT_int_double : (x:java_int)
      (RDT int double (int_to_mixed x) (int_to_mixed x))
  | RDT_raw_single : (x:java_short)
      (RDT raw single (short_to_mixed x) (short_to_mixed x))
  | RDT_raw_double : (y:java_int)
      (RDT raw double (short_to_mixed (pro y)) (int_to_mixed y)).
```

La séquence des relations `TDT` et `RDT` donne la relation de transformation entre les éléments des piles de la machine virtuelle *Java* et celles de la machine virtuelle *JavaCard*. On définit par simple induction sur la structure de liste ces relations pour des piles.

La traduction directe par les représentations La phase de traduction des instructions du code ne dépend que des représentations inférées sur les piles. En effet, les instructions émises ne manipulent que des quantités de représentation `double` (`java_int`) ou `single` (`java_short`) ; la distinction `ext` ou `raw` n'intervient pas. On définit donc une transformation des piles JVM vers les piles JCVM directement basée sur les représentations. Pour la représentation `double`, on a toujours l'identité :

$$In_{double} \ = \ id_{\mathbf{I}} \qquad Out_{double} \ = \ id_{\mathbf{I}}$$

Pour la représentation `single`, notre traduction doit rester correcte que le type soit `ext` ou `raw`, on prend donc l'intersection des deux relations In_{ext} (inj) et In_{raw} (pro^{-1}) pour In_{single}. De la même manière, on définit Out_{single} par la réunion des deux relations Out_{ext} (inj^{-1}) et Out_{raw} (pro).

$$In_{single} \ = \ inj \qquad Out_{single} \ = \ pro$$

On peut remarquer que ces relations sont en fait des fonctions totales. Cela sera très utile au moment des preuves. Il faut donc conserver cette information et disposer d'un lemme exprimant cette propriété.

3.4 La sémantique des instructions JVM et JCVM et la traduction

3.4.1 Les sémantiques JVM et JCVM

Nous définissons la sémantique opérationnelle d'un programme *Java* (ou *JavaCard*) vu comme un transformateur de piles, puis comme un transformateur de compteur ordinal. Les instructions JVM que l'on considère sont un sous-ensemble des instructions arithmétiques de Java, à savoir `dup`, `i2s`, `iadd`, `ifeq`, `ldc`. Pour la partie *JavaCard* on utilisera les variantes 16 bits et 32 bits des instructions précédentes ainsi que quelques autres comme `icmp` qui sont nécessaires pour permettre l'émission de code JCVM correct. La sémantique des instructions de chaque langage est décrite à l'aide de deux fonctions. La première, notée $[\![\ ]\!]$ et que nous appelons `semantics` transforme une pile (dont le sommet contient les paramètres de l'instruction en une nouvelle pile dont le sommet est le résultat). La seconde `branching` sert au calcul de la prochaine valeur du compteur ordinal en fonction du compteur courant et de la pile. Ces deux fonctions sont partielles, elles ne sont définies que sur les piles contenant suffisamment de paramètres ; on ne peut pas exécuter l'instruction `iadd` si la pile est vide alors que `iadd` attend deux éléments sur la pile.

Voici par exemple comment nous définissons la sémantique `semantics` des instructions de la machine JVM :

```
Inductive JVM_semantics : JVM_Instruction->JVM_stack->JVM_stack->Prop :=
  | JVMS_dup : (a:java_int;l:JVM_stack)
      (JVM_semantics JVM_dup (cons a l) (cons a (cons a l)))
  | JVMS_i2s : (a:java_int;l:JVM_stack)
      (JVM_semantics JVM_i2s (cons a l) (cons (inj (pro a)) l))
  | JVMS_iadd : (a:java_int;b:java_int;l:JVM_stack)
```

```
      (JVM_semantics JVM_iadd (cons a (cons b l)) (cons (iadd a b) l))
  | JVMS_ifeq : (pc:nat;a:java_int;l:JVM_stack)
      (JVM_semantics (JVM_ifeq pc) (cons a l) l)
  | JVMS_ldc :  (a:java_int;l:JVM_stack)
    (JVM_semantics (JVM_ldc a) l (cons a l)).
```

Regardons plus en détail ce qui se passe pour l'instruction `JVM_i2s` qui convertit un `java_int` en un `java_short` : `JVM_i2s` est exécutable si la pile contient au moins un élément a, et elle dépose sur la pile le résultat qui est $inj(pro\ a)$.

La sémantique des instructions *JavaCard* est définie de la même manière ; on définit aussi pour la JCVM la sémantique d'une séquence d'instructions de manière à pouvoir décrire l'exécution d'un bloc d'instructions, considéré comme *atomique* correspondant au même compteur de programme.

Les sémantiques de toutes les instructions JVM (resp. JCVM) sont définies dans [LY97] (resp. [Sun99]).

3.4.2 La traduction

Nous disposons maintenant de toutes les informations nécessaires à la spécification de l'émission du code. La traduction associe à chacune des instructions de la machine virtuelle *Java* une séquence d'instructions JCVM. En sortie, on autorise les instructions *JavaCard* qu'elles travaillent sur 16 ou 32 bits. Cette traduction ne dépend que de la représentation inférée pour les piles.

La fonction de traduction, notée $(I, r1, r2) \ \rightarrow < I >_{r1 \rightarrow r2}$, où I est l'instruction à traduire, et $r1 \rightarrow r2$ sa représentation ($r1$: représentation avant l'exécution de cette instruction et $r2$: représentation après) n'est définie que pour les représentations convenables des instructions. Si l'on considère l'exemple de `iadd`, les deux seules représentations autorisées sont :

$$\texttt{single} : \texttt{single} : \phi \rightarrow \texttt{single} : \phi \qquad \texttt{double} : \texttt{double} : \phi \rightarrow \texttt{double} : \phi$$

D'autres représentations, par exemple `double single` $\rightarrow$ `single` sont possibles. On n'autorise pas ce genre de représentations parce qu'elles conduisent à du code de mauvaise qualité (ici `swap_x 2,1 ; i2s ; sadd`).

Cette fonction de traduction est encore une fois représentée par un prédicat inductif en *Coq* dont on donne ici quelques extraits.

```
Inductive instruction_translation :
    JVM_Instruction -> instruction_repr -> (list JCVM_Instruction) -> Prop :=
  | IT_dup_single : (l:stack_representation)
      (instruction_translation JVM_dup ((cons single l),(cons single (cons single l)))
                               (cons JCVM_dup (nil JCVM_Instruction)))
  | IT_dup_double : (l:stack_representation)
      (instruction_translation JVM_dup ((cons double l),(cons double (cons double l)))
                               (cons JCVM_dup2 (nil JCVM_Instruction)))
...

  | IT_iadd_single : (l:stack_representation)
      (instruction_translation JVM_iadd ((cons single (cons single l)),(cons single l))
                               (cons JCVM_sadd (nil JCVM_Instruction)))
  | IT_iadd_double : (l:stack_representation)
      (instruction_translation JVM_iadd ((cons double (cons double l)),(cons double l))
                               (cons JCVM_iadd (nil JCVM_Instruction)))
...
```

```
| IT_ifeq_single : (l:stack_representation;pc:nat)
    (instruction_translation (JVM_ifeq pc) ((cons single l),l)
                             (cons (JCVM_ifeq pc) (nil JCVM_Instruction)))
| IT_ifeq_double : (l:stack_representation;pc:nat)
    (instruction_translation (JVM_ifeq pc) ((cons double l),l)
       (cons (JCVM_sipush zero_s) (cons JCVM_icmp (cons (JCVM_ifeq pc)
                                                  (nil JCVM_Instruction))))).
```

On peut remarquer que les traductions sont toujours très immédiates sauf peut-être pour l'instruction `ifeq` et la représentation *double* : $\phi \rightarrow \phi$. Dans ce cas, la séquence émise consiste en : le chargement de la valeur zéro (16 bits) que l'on étend à 32 bits (`sipush`), la comparaison des 2 valeurs du sommet de la pile (`icmp`), puis la décision de saut à partir du résultat de la comparaison (`ifeq`). Il est intéressant de remarquer que cette définition correspond au code de l'émission de bytecode en fonction des résultats de l'analyse statique. C'est l'une des rares parties informatives des définitions qui ont un contenu calculatoire, plutôt que d'exprimer de simples propriétés logiques.

4 Propriétés démontrées

Dans cette partie, nous présentons les résultats démontrés à l'aide de l'assistant *Coq*. Nous commençons par présenter quelques lemmes utiles tout au long des preuves. Dans un deuxième temps, nous présentons une preuve de correction de l'émission du bytecode JCVM. Nous terminons par une preuve que le programme *JavaCard* traduit simule bien le programme *Java* de départ.

4.1 Quelques lemmes intéressants

Les relations fonctionnelles Notre méthode de modélisation des fonctions partielles sous forme de relations cache parfois des informations fondamentales, comme la fonctionnalité de certaines de ces relations par exemple. Pour toutes les relations représentant des fonctions partielles, on doit garder à l'esprit le fait que l'on manipule des fonctions. On définit donc un prédicat `functional` et on montre que certaines relations comme `JVM_semantics` sont effectivement bien fonctionnelles.

```
Definition functional := [A:Set;B:Set;R:A->B->Prop]
           (x:A;y,z:B)(R x y) -> (R x z) -> y=z.
Syntactic Definition Functional := (functional ? ?).
```

Les fonctions totales Dans notre preuve de simulation, nous aurons besoin de construire des points intermédiaires sur le diagramme décrivant la simulation. Pour cela, il faudra avoir montré que certaines fonctions, toujours représentées sous forme de relations, sont totales sur un domaine considéré (domaine défini par un prédicat d'appartenance). Soit f une fonction et P sa représentation relationnelle ; on dit que f est totale sur C si et seulement si :

$$\forall P : A \rightarrow B, x : A, \qquad x \in C \rightarrow \exists y : B \mid (P\ x\ y)$$

Factorisation des preuves d'existence Finalement, il est utile de définir quelques lemmes exprimant des propriétés telles que le fait que la séquence $In_r; Out_r$ avec r une représentation est l'identité. Un tel lemme fournit une preuve d'existence d'une pile JVM (le témoin à donner pour prouver la séquence) correspondant à une pile JCVM quelconque. Sous réserve de bonne formation des représentations par rapport aux données - les deux listes sont de même taille et la représentation `double` (resp. `single`) correspond à un élément `java_int` (resp. `java_short`) -, on montre par le lemme suivant qu'il existe une liste d'entiers 32 bits (une pile pour la JVM) correspondant à la pile JCVM donnée.

```
Lemma lemma_inout_stack : (lr : stack_representation)
                          (lm : JCVM_stack)(good_repr_stack lr lm) ->
                          ((Seq (stack_IN lr) (stack_OUT lr)) lm lm).
```

Cela permet de factoriser les preuves constructives d'existence de piles qui se font nécessairement par induction sur la structure de liste. On peut utiliser ce genre de lemme dans une grande preuve pour éviter d'avoir à reconstruire explicitement une pile pour en prouver l'existence.

4.2 Correction de la traduction

On procède instruction par instruction, en considérant les différentes représentations pour lesquelles la traduction a été définie. Pour une instruction donnée associée à une représentation (`r1`,`r2`) donnée, on doit montrer la propriété suivante :

$$[\![< I >_{r1 \to r2}]\!] \ = \ In_{r1} \ ; \ [\![I]\!] \ ; \ Out_{r2}$$

où $< \ I \ >_{r1 \to r2}$ est la traduction de l'instruction I définie dans la section 3.4.2.

Par exemple pour l'instruction `iadd` ayant le type `single single` $\to$ `single`, on doit montrer que :

$$[\![sadd]\!] \ = \ in \ \mathtt{x} \ in \ ; \ [\![iadd]\!] \ ; \ pro$$

sur le domaine $\mathbf{S}^2$. Il s'agit donc de prouver que deux fonctions (de domaines finis et égaux) sont égales. On pourrait craindre d'avoir à montrer cette propriété par extension ; cependant il ne sera jamais nécessaire de raisonner séparément sur les éléments du domaine. L'axiome `iadd_sadd` nous permet de conclure la preuve sans difficulté. Le théorème que nous avons établi en *Coq* est le suivant :

```
Theorem correctness :
  (I:JVM_Instruction)(r1:(list representation))
  (r2:(list representation))(l:(list JCVM_Instruction))
     (x,y:JCVM_stack)
         (instruction_translation I (r1,r2) l) ->
         (good_repr_stack r1 x) ->
         ((JCVM_seq_semantics l x y) <->
                          ((Seq (Seq (stack_IN r1) (JVM_semantics I))
                                (stack_OUT r2)) x y)).
```

La première hypothèse dont on dispose est que l est la traduction de I pour la représentation $r1 \to r2$. On dispose également d'une propriété de bonne formation sur $r1$ et x ; en effet, la pile x des valeurs JCVM doit avoir le même nombre d'éléments que

la pile des représentations $r1$. De plus les valeurs et les représentations doivent être compatibles : une valeur codée sur 32 bits (respectivement 16 bits) a nécessairement la représentation `double` (respectivement `single`).
Prouver la correction de la traduction revient à montrer sur le schéma suivant que :

1. Connaissant le chemin en "trait plein", on peut déduire le chemin en "pointillé" (passage de la JCVM à la JVM).
2. Connaissant le chemin "pointillé", on peut en déduire le chemin en "trait plein".

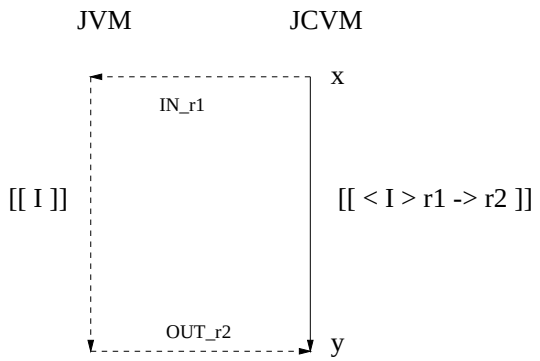

FIG. 1 – Propriété de correction

Pour montrer une telle propriété, on procède par cas sur l'instruction du bytecode JVM à traduire. Ensuite, grâce à notre choix de définir les fonctions partielles sous forme de relations et toutes les relations sous forme de prédicats inductifs, on peut utiliser les tactiques d'*Inversion* de façon à construire les formes des piles x et y pour lesquels le contexte n'est pas inconsistant. En effet pour chaque sens de l'équivalence, on dispose d'une hypothèse sur la sémantique (obtenue directement pour la machine *JavaCard* ou indirectement par l'intermédiaire des séquences pour la machine *Java*). On peut ensuite généralement conclure en utilisant les lemmes démontrés dans la section 4.1 ou les définitions inductives des prédicats ainsi que les tactiques de base de Coq `Rewrite`, `Reflexivity`...

4.3 Simulation

4.3.1 Correction du typage

Il faut commencer par définir les propriétés de correction du typage : On dit que f, une fonction des piles JVM dans les piles JVM, admet le type σ, noté $f : \sigma$ avec $\sigma = \varsigma_1 \rightarrow \varsigma_2$ lorsqu'elle vérifie les deux conditions suivantes [2] :

- (f ; Out_{ς_2}) est totale sur l'image de In_{ς_1}
- (In_{ς_1} ; f ; Out_{ς_2}) est fonctionnelle.

Pour bien comprendre ce que cela signifie, nous allons regarder quels types sont admissibles pour l'instruction `iadd`. Intuitivement, le type `ext ext` $\rightarrow$ `ext` ne convient pas ; l'addition de deux entiers 16 bits peut en effet conduire à un débordement de capacité, et dans ce cas le résultat obtenu ne donnera que les 16 bits de poids faible du résultat réel. Par contre, le type `raw raw` $\rightarrow$ `raw` convient parfaitement, si le mot de poids fort du résultat n'est pas utile dans la suite des calculs, les mots de poids fort

[2] " ;" dénote la séquence de relations

des 2 paramètres ne le sont pas non plus. Regardons comment cela se traduit dans la définition formelle précédente :

- `iadd` admet-elle le type `ext ext` $\rightarrow$ `ext` ?
  La première condition exige que (iadd ; $Out_{\texttt{ext}}$) soit totale sur l'image de $In_{\texttt{ext ext}}$. Comme $In_{\texttt{ext ext}} = inj$ `x` inj et $Out_{\texttt{ext}} = inj^{-1}$, cela signifie qu'il faut que (iadd ; inj^{-1}) soit totale sur l'image de inj `x` inj. Or, il est clair que l'addition de deux éléments de $inj(\mathbb{S})$ peut déborder et donc donner un résultat dont le mot de poids fort ne sera pas redonnant avec le bit 15 du mot de poids faible. Il sera donc impossible d'appliquer inj^{-1} au résultat. La fonction n'est donc pas totale et par conséquent `ext ext` $\rightarrow$ `ext` n'est pas un typage correct de l'instruction `iadd`.
- `iadd` admet-elle le type `raw raw` $\rightarrow$ `raw` ?
  La première condition exige que (iadd ; $Out_{\texttt{raw}}$) soit totale sur l'image de $In_{\texttt{raw raw}}$. Comme $In_{\texttt{raw raw}} = pro^{-1}$ `x` pro^{-1} et $Out_{\texttt{raw}} = pro$, cela signifie qu'il faut que (iadd ; pro) soit totale sur l'image de pro^{-1} `x` pro^{-1}. L'image de pro^{-1} `x` pro^{-1} est $\mathbb{I}$ `x` $\mathbb{I}$. Comme `iadd` est totale sur $\mathbb{I}$ `x` $\mathbb{I}$ et que `pro` est totale sur $\mathbb{I}$, la première condition est bien vérifiée. Il faut maintenant vérifier que ($In_{\texttt{raw raw}}$; iadd ; $Out_{\texttt{raw}}$) c'est-à-dire (pro^{-1} `x` pro^{-1} ; iadd ; pro) est bien fonctionnelle. $In_{\texttt{raw raw}}$ réinvente de l'information sur les mots (16 bits) de poids fort des opérandes. Cependant, comme on ne s'intéresse qu'aux 16 bits de poids faible du résultat, les données réinventées n'affecteront pas le calcul de ces 16 bits (qui sont récupérés par projection `pro` du résultat de `iadd`).

La propriété de totalité permet de montrer qu'on peut faire un pas de calcul dans la JCVM, alors que la fonctionnalité de la séquence assure que le programme aura le même comportement quelle que soit les valeurs données aux bits *non-pertinents* réinventés lors des transformations d'éléments de type `java_short` (typé `raw`) dans la pile JCVM vers des entiers 32 bits de la pile JVM.

4.3.2 La preuve de la simulation

Il s'agit maintenant de montrer que le programme *JavaCard* obtenu simule bien le programme *Java* de départ si les hypothèses de correction du typage sont vérifiées. On cherche à montrer que le programme JCVM obtenu par l'algorithme de traduction a bien le même comportement observable que le programme JVM de départ.

L'idée intuitive de la simulation est que si la machine JVM "avance d'un pas" dans l'exécution du programme (i.e., exécute une instruction), alors la machine JCVM doit pouvoir avancer, elle aussi, d'un pas dans l'exécution du programme JCVM correspondant. De plus, si les piles des programmes JVM et JCVM, avant ce pas de calcul, étaient reliées par la relation de correspondance définie dans la section 3.3.2, alors les piles obtenues à l'issue de l'exécution de l'instruction dans la machine JVM et de sa traduction dans la machine JCVM doivent rester en correspondance par cette même relation. En résumé, la relation de simulation exprime le fait que le programme JCVM calcule bien la même chose que le programme JVM, à chaque étape de l'exécution de ce programme.

Définition de la simulation Si le programme *Java* et son homologue traduit *JavaCard* sont arrivés en des états respectifs de la pile sf et msf de type $t1$ et de représentation $r1$ vérifiant :

$$((Out_{t1}\ ;\ \texttt{RDT}\ t1\ r1)\ \ sf\ \ msf)$$

et si I est l'instruction *Java* suivante de type $t1 \rightarrow t2$ et de représentation $r1 \rightarrow r2$) et que l'on sait que

$$[\![I]\!]\ sf = sf',$$

alors il existe une pile $msf4$ reliée à sf' par

$$((Out_{t2}\ ;\ \texttt{RDT}\ t2\ r2)\ \ sf'\ \ msf4)$$

et vérifiant

$$[\![Iseq]\!]\ msf = msf4.$$

Le théorème énoncé en *Coq* est le suivant :

```
Theorem simulation :
  (I:JVM_Instruction;I_seq:(list JCVM_Instruction);
   sf,sf':JVM_stack;msf:JCVM_stack;
   r1,r2:stack_representation;t1,t2:(list type))

    (Total (In_range (stack_TDT' t1)) (Seq (JVM_semantics I) (stack_TDT t2))) ->
    (Functional (Seq (Seq (stack_TDT' t1) (JVM_semantics I)) (stack_TDT t2))) ->
    (stack_translation t1 r1 sf msf) ->
    (instruction_translation I (r1,r2) I_seq) ->
    (JVM_semantics I sf sf') ->
    (stack_type_representation t1 r1) ->
    (stack_type_representation t2 r2) ->
    (Ex [msf':JCVM_stack]
       ((stack_translation t2 r2 sf' msf') /\ (JCVM_seq_semantics I_seq msf msf'))).
```

La figure 4.3.2 présente le diagramme que l'on cherche à établir, les flèches solides sont les hypothèses du problème et on cherche à construire le point $msf4$.

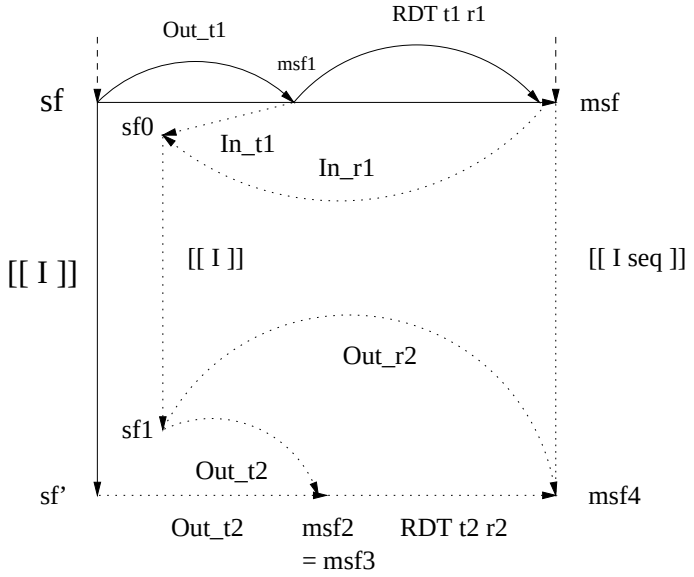

FIG. 2 – La simulation du programme JVM par le programme JCVM

Preuve Nous allons décrire les principales étapes de la preuve de cette propriété :

1. On commence par construire le point $msf1$ qui est l'élément intermédiaire de la séquence (Out_{t1} ; `RDT` $t1$ $r1$).
2. On exhibe maintenant $sf0$ qui est obtenu par l'application de In_{r1} à msf ; On utilise alors le théorème exprimant que (RDT $t1$ $r1$; In_{r1}) $\subset$ In_t1 que nous avons prouvé par ailleurs pour relier $msf1$ à $sf0$.
3. $sf0$ est dans l'image de In_{t1} ; par l'hypothèse de typage correct (première condition), on obtient donc les points $sf1$ et $msf2$.
4. Sachant que In_{t1} et Out_{t1} sont réciproques l'une de l'autre, on a le fait que sf est aussi dans l'image de In_{t1}. En utilisant la première règle de la propriété "typage correct", on obtient deux nouveaux points sf' et $msf3$.
5. En utilisant la fonctionnalité de In_{t1} ; $[\![I]\!]$; Out_{t2} (seconde propriété du typage correct), on en déduit que $msf3$ et $msf2$ sont égaux.
6. A partir de $sf1$, on construit un nouveau point $msf4$ par Out_{r2}. On utilise ici le fait que Out_{r2} est une fonction totale.
7. Comme on a (Out_{t2} $sf1$ $msf2$) et (Out_{r2} $sf1$ $msf4$), on en déduit que (`RDT` $t2$ $r2$ $msf2$ $msf4$), en utilisant un lemme faisant intervenir les relations entre types et représentations.
8. Ensuite, comme on a deux relations (Out_{t2} sf' $msf2$) et (RDT $t2$ $r2$ $msf2$ $msf4$) qui sont les composantes de la relation de traduction des piles. On a donc bien la relation attendue entre les piles JVM sf' et JCVM $msf4$.
9. Finalement, comme par hypothèse ($[\![\ I\_seq\ ]\!]$ = In_{r1} ; $[\![\ I\ ]\!]$; Out_{r2}), msf et $msf4$ sont bien reliées par la relation $[\![\ I\_seq\ ]\!]$.

5 Conclusions et perspectives

Ce stage a permis de montrer que la partie "émission de code" du traducteur est bien correct sous réserve que l'inférence des types réalisée au préalable vérifie certaines hypothèses. On a de plus montré que le programme JCVM émis simule bien le programme JVM initial. La simulation a été montrée en s'appuyant sur le théorème de correction de la traduction. Ainsi, il est indépendant des instructions considérées. Si l'on désire augmenter le nombre d'instructions mises en jeu, il suffirait de montrer le cas correspondant dans le théorème de correction de la traduction.
Les perspectives de développement ultérieures sont nombreuses ; il faudrait commencer par ajouter les propriétés relatives aux compteurs de programme et aux branchements dans les preuves de correction de la traduction et de simulation. Dans un deuxième temps, il serait intéressant de montrer que la phase d'analyse statique du code JVM est correct. On pourrait aussi envisager de montrer une relation de bi-simulation entre les programmes *Java* et *JavaCard*. De cette manière, on aurait des résultats sur le comportement de la machine virtuelle JCVM lors d'un plantage de la machine JVM. Cependant, le code considéré en entrée est supposé passer avec succès le test du "bytecode verifier". On devra donc avoir l'assurance que la JVM ne plantera pas, mais comment peut-on avoir une telle assurance tant que le vérificateur de bytecode n'a pas été prouvé formellement ?

Pour ce qui concerne la modélisation en *Coq*, il sera très utile d'introduire de la syntaxe pour représenter les séquences de relations par " ;" par exemple. De plus, l'utilisation des arguments implicites pourraient permettre d'augmenter la lisibilité des développements. On se trouve parfois en cours de preuve avec un but contenant plus

d'arguments implicites affichées que des données utiles à sa résolution.
Un des avantages du système *Coq* est qu'il permet la vérification automatique des preuves (`type-checking`) des théorèmes démontrés par interaction avec l'utilisateur. Cela donne une garantie forte de correction de ce qui a été prouvé. On doit pouvoir facilement convaincre quelqu'un de la correction des lemmes énoncés en montrant simplement que *Coq* les accepte. Cette caractéristique est essentielle ; elle permet de présenter ces résultats à des personnes qui n'ont pas de compétence particulière en logique. Pour se convaincre de la validité d'un théorème il suffit d'en faire vérifier la preuve par *Coq* plutôt que de devoir se plonger dans le langage de tactiques et d'étudier les scripts de preuve.

Références

[BBC+99] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, P. Loiseleur, C. Muñoz, C̃. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.3. Technical report, INRIA, July 1999. Version révisée distribuée avec Coq.

[Gim98] E. Giménez. A tutorial on recursive types in coq. Rapport technique 221, INRIA, May 1998. disponible à http ://coq.inria.fr/doc-eng.html.

[HKPM99] G. Huet, G. Kahn, and Ch. Paulin-Mohring. The Coq proof assistant - a tutorial, version 6.2.4. Rapport technique, INRIA, January 1999. Version révisée distribuée avec Coq.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[Pot99] François Pottier. Présentation du traducteur optimisant de Java vers JavaCard. 1999. Document interne Trusted Logic.

[Sun99] Sun microsystems Inc. *JavaCard 2.1 Virtual Machine Specification*. 1999. disponible à http ://java.sun.com/products/javacard/javacard21.html.