Rapport de stage - DEA SPP

Guillaume DUFAY

Septembre 2000

Stage de DEA effectué à l'INRIA Sophia-Antipolis sous la direction de Gilles BARTHE, Chargé de Recherches.

Ce stage a été réalisé dans le cadre du Diplôme d'Études Approfondies *Programmation : Sémantique, Preuves et Langages* cohabilité par l'Université Denis Diderot (Paris VII), l'Université Pierre et Marie Curie (Paris VI), l'Université Paris Sud (Paris XI), l'École Normale Supérieure de Paris, l'École Normale Supérieure de Cachan, l'École Polytechnique et le Conservatoire National des Arts et Métiers.

Il s'est déroulé sous la direction de Gilles Barthe à l'Institut National de Recherche en Informatique et Automatique (inria) de Sophia-Antipolis dans l'équipe oasis (Objets Actifs, Sémantique, Internet et Sécurité), dirigée par Isabelle Attali.

Remerciements particuliers à Line Jakubiec, Bernard Serpette, Simão Sousa pour leur aide et conseils.

Résumé

Face aux impératifs de sécurité réclamés par les applications destinées aux cartes à puces intelligentes, les méthodes formelles apportent une solution adaptée. Nous nous intéressons plus particulièrement au langage *Java Card* et nous nous proposons alors de formaliser ce langage dans l'assistant de preuves COQ. Cette formalisation constitue le point de départ de nombreux développements dans la construction d'un environnement certifié pour le langage *Java Card*.

Mots-clé : JavaCard, Sécurité, Coq, Smart Card, Méthodes formelles, Vérificateur de bytecode

Abstract

Toward the requirements of security claimed by the applications intended for the smart cards, formal methods bring an adapted solution. We are interested more particularly in the language *Java Card* and then we propose to formalize this language in the proof assistant COQ. This formalization constitutes the starting point of many developments in the construction of a certified environment for the language *Java Card*.

Keywords: JavaCard, Security, Coq, Smart Card, Formal Methods, Bytecode verifier

Table des matières

1 Introduction **1**

2 Le langage *Java Card* **1**
- 2.1 Présentation générale . . . 1
- 2.2 Spécificités . . . 2
- 2.3 Compilation . . . 3
- 2.4 Sécurité . . . 3

3 L'assistant de preuves COQ **4**
- 3.1 Formalisme . . . 4
- 3.2 Utilisation . . . 5

4 Formalisation en COQ de *Java Card* **5**
- 4.1 Formalisation des structures de données . . . 5
- 4.2 Instructions . . . 12
- 4.3 Détails d'une instruction . . . 12
- 4.4 Exécution . . . 15

5 L'abstraction de types **15**
- 5.1 Modifications apportées à la machine virtuelle . . . 16
- 5.2 Détails d'une instruction abstraite . . . 16
- 5.3 Preuve de correction de l'abstraction . . . 18

6 Le vérificateur de bytecode **19**
- 6.1 Les fonctionnalités d'un vérificateur de bytecode . . . 19
- 6.2 Algorithme . . . 20
- 6.3 Preuve de terminaison . . . 21
- 6.4 Bonne fondaison de l'ordre sur le treillis . . . 24
- 6.5 Implémentation . . . 25

7 Conclusion **27**
- 7.1 Travaux connexes . . . 27
- 7.2 Développements futurs . . . 28

A Opérateur d'unification **31**

1 Introduction

Depuis le lancement des cartes à puce en 1980 par les PTT, leur rôle ne se limite plus à la détention d'informations, elles ont gagné une certaine intelligence par la capacité d'exécuter des programmes, d'où le nom de *smart cards* donné à la dernière génération de cartes à puce.

En l'absence d'un langage de programmation dédié aux *smart cards*, la programmation de ces cartes à puce s'effectuait à l'aide d'un assembleur particulier à la puce utilisée. L'introduction du langage *Java Card*, une version simplifiée et dédiée aux cartes à puce du populaire langage *Java*, a permis à une plus large majorité de développeurs de créer des applications pour *smart cards*, lesquelles applications pouvant alors fonctionner sur toute carte à puce implémentant une machine virtuelle *Java Card*.

Face aux impératifs de sécurité réclamés pour les utilisations des *smart cards* (comme la confidentialité, l'authentification, l'intégrité des données stockées sur la carte), le langage *Java Card* apporte des solutions spécifiques, comme le mécanisme de *firewall*. De plus, en raison de l'absence de certaines fonctionnalités évoluées de *Java* dans *Java Card*, ce langage se prête mieux que *Java* à la vérification formelle de propriétés sur la plate-forme *Java Card* elle-même ou bien sur des programme.

Ainsi, pour établir de manière formelle des propriétés de sécurité, la machine virtuelle *Java Card* et son environnement d'exécution (Java Card Virtual Machine, JCVM, et Java Card Runtime Environment, JCRE, respectivement) ont été formalisés dans l'assistant de preuves COQ. Cette formalisation laisse ainsi à disposition la possibilité de prouver des propriétés sur la plate-forme formalisée elle-même ou sur de programmes *Java Card*, en les exécutant sur cette plate-forme dans l'environnement COQ.

Seule une telle formalisation, imposée pour atteindre les plus hauts niveaux définis par les procédés des certification tels que *Common Criteria* ou *ITSEC*, peut apporter la confiance requise pour l'utilisation massive des *smart cards*.

2 Le langage *Java Card*

2.1 Présentation générale

Java Card est une version allégée du langage *Java* conçue pour les *smart cards*, dont les capacités mémoire et calcul sont réduites. L'exécution d'un programme *Java Card* est rendue possible par la présence sur la carte d'une machine virtuelle (la JCVM, détaillée dans [12]) et de son environnement d'exécution (le JCRE, détaillé dans [11]), qui peut-être vu comme une sorte de système d'exploitation.

La carte à puce n'a pas de fonctionnement autonome, elle est dépendante pour l'exécution des programmes qu'elle contient d'un lecteur de carte appelé *CAD* (Card Acceptance Device). Elle communique avec ce lecteur par l'intermédiaire d'un protocole nommé *APDU* (Application Protocol Data Unit).

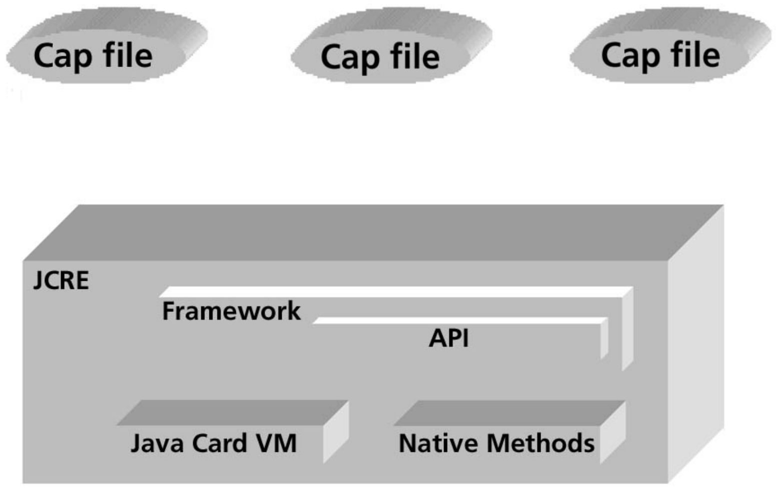

FIG. 1 – *Architecture générale de l'environnement* Java Card.

2.2 Spécificités

La syntaxe du langage reste la même par rapport à *Java* mais certaines fonctionnalités comme le traitement des nombres réels ou les mécanismes de *garbage collection*, de *class loader* ou de *threads* ne sont pas prévues à cause des capacités limitées de l'électronique intégrée à la carte. Cependant, l'absence de certaines de ces fonctionnalités présente des avantages pour ce qui a trait à la sécurité.

Le chargement dynamique de classes est par exemple une source majeure d'insécurité [8] dans le langage *Java*. Si ce mécanisme est absent de *Java Card*, il est toutefois possible de charger sur la carte, après sa mise en service, des programmes *Java Card*, des *applets* suivant une terminologie commune à *Java*, mais cette opération est impossible durant l'exécution de programmes.

L'installation d'applets chargés sur la carte se fait ensuite par l'intermédiaire de protocoles cryptographiques à clefs secrètes assurant ainsi que seuls des applets d'origine connue seront exécutés.

L'absence de *threads* dans *Java Card* élimine la possibilité pour un applet d'interférer ou d'observer l'exécution d'autres applets. Cela apporte également un certain déterminisme dans l'exécution des applets et simplifie en conséquence les preuves de propriétés de ces programmes.

Enfin, par rapport à *Java* et pour des raisons de sécurité décrites dans 2.4, on associe à chaque objet un *contexte*. Ce contexte décrit le package, l'ensemble d'applets, dans lequel l'objet a été défini.

2.3 Compilation

La génération d'un programme *Java Card* s'effectue en plusieurs étapes. Le programme est d'abord écrit dans un fichier portant l'extension `.java` puis compilé en un *class file* par un compilateur standard *Java*. Ce fichier `.class` est ensuite converti en un fichier `.cap` (Converted APplet) par un programme appelé *converter* qui s'assure que le code écrit est bien du code *Java Card* et optimise les données du programme pour son installation sur une carte à puce.

Des outils de simulation sont disponibles pour observer les communications entre la carte à puce et un lecteur de cartes.

2.4 Sécurité

Les contextes En raison de la présence simultanée sur une carte de plusieurs applets, il a fallu prévoir un environnement sûr à l'intérieur duquel peuvent s'exécuter des applets. Le JCRE a pour rôle de gérer et de garantir cet environnement.

Grâce à la notion de contexte et par un mécanisme dit de *firewall*, le JCRE va empêcher toute communication entre applets de contextes différents, c'est-à-dire entre applets définis dans des packages différents.

Lorsqu'un nouvel objet est créé (instruction `new`), le contexte courant (le contexte de l'objet auquel appartient la méthode actuellement exécutée) est associé à l'objet créé. Lors de tout appel de méthode ou accès de variable, le JCRE va comparer le contexte de l'objet auquel on veut accéder au contexte courant. Il existe aussi un contexte particulier, celui du JCRE, disposant des accès à tous les objets.

Néanmoins, ce modèle est beaucoup trop restrictif et la communication entre applets est parfois nécessaire. Celle-ci est alors rendue possible par les concepts de points d'entrée, de tableaux globaux et d'objets partagés.

Les points d'entrée et tableaux globaux Les points d'entrée et les tableaux globaux sont des attributs de méthode et de tableaux respectivement. Ces attributs sont fixés par le JCRE lors de l'appel d'une méthode système.

Si une méthode a été déclarée comme point d'entrée, on peut alors l'appeler de n'importe quel contexte et passer ainsi outre les restrictions du *firewall*. Cet appel de méthode produit un changement de contexte courant vers la méthode appelée.

Les tableaux déclarés comme globaux voient leurs champs accessibles depuis n'importe quel contexte. Le buffer d'APDU, stockant les transactions avec un lecteur de carte, est ainsi par exemple déclarée comme un tableau global pour permettre à tout applet d'y accéder.

Ces deux types d'attributs pouvant être temporaire, il faut éviter qu'une référence vers un ces objets ne soit stockée dans une variable ou un tableau. En effet, l'accès direct par la référence stockée contournerait alors le *firewall*.

Si un accès incorrect à un objet est réalisé ou si une référence à un objet privilégié est stockée, le JCRE intervient et lance une exception de sécurité.

Les objets partagés Les objets partagés (*shareable object*) sont des objets implémentant l'interface `Shareable`. Il est possible de partager de tels objets entre plusieurs contextes.

Le mécanisme pour obtenir un objet partagé s'effectue par un transaction avec l'intermédiaire du JCRE. L'objet implémentant `Shareable` peut refuser de fournir un objet partagé suivant l'identifiant du demandeur. L'appel de méthodes de l'objet partagé par l'objet ayant fait la requête conduit aussi à un changement de contexte.

Ainsi, la charge du changement de contexte est reportée avec ce mécanisme du JCRE vers les objets implémentant `Shareable`.

La sécurité en COQ Le but de la formalisation en COQ de la machine virtuelle *Java Card* est d'étudier la sécurité du langage à deux niveaux :

- Au niveau de la plate-forme : On se concentrera ici sur les propriétés de sûreté du langage, en particulier la sûreté du typage et des propriétés liées à la gestion de la mémoire, en utilisant des assistants de preuve, bien adaptés au support de la formalisation et au raisonnement logique ;
- Au niveau applicatif : Il s'agit ici de montrer qu'un programme donné se conforme à une politique établie de sécurité en utilisant des outils variés tels que des assistants de preuve ou des *model checkers*.

3 L'assistant de preuves COQ

3.1 Formalisme

Le langage formel de COQ [1] repose sur le *Calcul des Constructions Inductives* développé par Thierry COQUAND, Gérard HUET et Christine PAULIN. Il s'agit d'un λ-calcul d'ordre supérieur basé sur l'isomorphisme de Curry-Howard.

COQ permet de manier des types dépendants, des constructions inductives et co-inductives et offre ainsi une très forte expressivité. D'autres part, des mécanismes de coercion implicite ou de paramètres implicites facilitent l'écriture de termes.

Une particularité intéressante de COQ est de pouvoir mêler dans un même terme des parties calculatoires et des parties propositionnelles. Il est alors possible d'extraire le contenu calculatoire vers des termes de F_ω. Une extraction vers *Objective CAML*, qui supporte la plupart des termes de F_ω, est par exemple implémentée dans COQ.

3.2 Utilisation

Pour la formalisation des instructions de la machine virtuelle *Java Card*, deux stratégies étaient envisageables dans COQ. Ces instructions pouvaient être écrites :

- ou bien en utilisant COQ comme un langage fonctionnel $f : A \rightarrow B$;
- ou bien en utilisant des relations inductives $R_f : A \rightarrow B \rightarrow Prop$.

La première de ces approches a été choisie, elle présente de nombreux avantages :

1. elle peut être aisément rapprochée et comparée à une implémentation classique ;
2. la formalisation se trouve ainsi sous une forme directement exécutable et des tests peuvent alors être effectués pour vérifier son adéquation aux spécifications ;
3. directement intégrée à l'outil de démonstration, la formalisation COQ est ainsi bien adaptée à la vérification de propriétés ;
4. enfin, les possibilités d'extraction vers les langages tels que *Objective Caml* ou *Java* (pour profiter de la rapidité d'exécution ou des traits impératifs de ces langages) ou bien vers d'autres assistants de preuves tels que *Isabelle* ou *HOL* apportent une grande portabilité à la formalisation

Nous étudions actuellement une stratégie pour passer automatiquement d'une spécification en COQ sous forme de langage fonctionnel en relations inductives, permettant ainsi de bénéficier des avantages deux formes de formalisation. Il semble en effet que pour certaines preuves, la formalisation sous forme de relations inductives soit plus intéressante car on peut alors utiliser les tactiques d'*inversion* de COQ.

4 Formalisation en COQ du langage *Java Card*

La formalisation en COQ du langage *Java Card* requiert la formalisation de la machine virtuelle et de l'environnement d'exécution *Java Card*. Pour cela, il faut au minimum formaliser le contenu du fichier CAP, la mémoire et le jeu d'instructions de la machine virtuelle.

4.1 Formalisation des structures de données

4.1.1 Type des données et valeurs

Le système de types La machine virtuelle *Java Card* distingue les types dits *primitifs* et les types de *références* :

- Les types primitifs englobent les types des valeurs arithmétiques (`Byte`, `Short`, `Int`), le type `Boolean`, le type `ReturnAddress` (utilisé pour les adresses de retour après l'instruction `JSR`). Pour définir ces types primitifs en COQ, on utilise une définition inductive dont chacun des constructeurs, sans arguments, représente un des types primitifs :

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

– Les types des références sont utilisés pour les types des instances de classe, des interfaces, et des tableaux. De la même façon que pour les types primitifs, on utilise une définition inductive pour représenter les types des références, cependant les constructeurs prennent ici des arguments. Ainsi, les instances de classes et les interfaces sont indexées, pour les identifier et les distinguer, par un entier (en raison de l'absence des chaînes de caractères dans COQ). Le constructeur du type des tableaux prend quant à lui comme argument le type des éléments du tableau, ce type pouvant être n'importe quel type de la machine virtuelle, un type de référence y compris.
  Cette spécificité pour le type des tableaux oblige à définir simultanément les types de la machine virtuelle (une définition inductive avec un constructeur pour les types primitifs et un autre pour les types des références), utilisant le type des références, et les types des références, utilisant les types de la machine virtuelle, par l'intermédiaire d'une définition inductive mutuelle :

```
Mutual Inductive type : Set :=
  Prim : type_prim -> type |
  Ref : type_ref -> type
with type_ref : Set :=
  Ref_array : type -> type_ref |
  Ref_instance : nat -> type_ref |
  Ref_interface : nat -> type_ref.
```

On remarquera que le système de types permet de typer les tableaux de tableaux, autorisés par *Java* mais pas par *Java Card* (principalement pour des raisons de mémoire). Cependant, il est facile de prouver que la formalisation, en particulier l'implémentation de la fonction de création de tableau (instruction `anewarray`) ne permet pas d'obtenir un tel type.

Les valeurs On souhaite ensuite utiliser un type de valeurs COQ permettant de stocker facilement n'importe quel valeur de type du système de types de la machine virtuelle précédemment défini. Le type COQ `nat` des entiers naturels n'est par exemple pas adapté à stocker des valeurs signées comme `Int`. Le type `Z` des entiers relatifs de COQ, codés en binaire, est lui suffisamment souple pour supporter des données comme `Int`, ou bien par l'intermédiaire d'une conversion implicite des données non calculatoires de type entier naturel des pointeurs d'instruction (`ReturnAddress`) ou des adresses mémoire. De plus, un entier de type `Z` est naturellement affiché sous forme d'entier par COQ (et non pas avec le nom de ses constructeurs). Le type `Z` sera donc utilisé pour stocker les valeurs de la machine virtuelle.

Le type `valu`, type des valeurs de la machine virtuelle s'écrit donc simplement comme le produit cartésien d'un type de la machine virtuelle et d'une valeur :

```
Definition valu := type*Z.
```

Du point de vue calculatoire, les calculs sur `Z` sont bien plus rapides que `nat` car ils sont effectués en binaire et non en unaire. En effet, `Z` est basé sur un type inductif `positive` avec les constructeurs :
`xI : positive -> positive;`
`xO : positive -> positive;`
`xH : positive`
où `xH` a la même signification que `xI` et termine un terme de type `positive`. Ainsi, le type `positive` est utilisé pour représenter des entiers strictement positifs. `Z` est alors naturellement défini avec les constructeurs :
`ZERO : Z`;
`POS : positive -> Z` pour les nombres positifs ;
`NEG : positive -> Z` pour les nombres négatifs.

Avec ce codage, chaque nombre entier a une unique représentation dans `Z`, ce qui permet d'utiliser l'égalité de LEIBNIZ. Cependant les opérations nécessaires à la machine virtuelle telles que la division euclidienne sont bien plus difficiles à écrire et ne sont pas présentes dans la librairie standard de COQ. De plus, les opérations sur ces entiers de COQ ne se conforment pas aux spécification de Sun des dépassements de capacité (*overflows*). Pour ces raisons, il a été développé une autre représentation binaire des entiers, nommée `Z_bits`.

Par une conversion entre `Z_bits` et `Z`, on peut fournir une division euclidienne dans `Z`.

Pour définir `Z_bits`, on construit un premier type inductif, `bit`, constitué des constructeurs `bitO` et `bitI`, puis `bits`, une liste de `bit`. On obtient ainsi une représentation conventionnelle des entiers, avec une interprétation choisie en mode *little-endian*. Contrairement à la représentation native des entiers COQ, le représentation d'un entier n'est pas unique (plusieurs `bitO` peuvent être ajoutés à la fin), cela simplifie les opérations sur ces nombres et se rapproche d'une représentation ordinaire des nombres binaires.

À partir de cette représentation des entiers naturels, on définit le type `Z_bits`, représentant les entiers signés, comme à nouveau une liste de `bit`. Simplement, le type `Z_bits` sera paramétré par un entier naturel indiquant la taille en bits des nombres traités, et interprété en complément à deux.

Cette représentation en complément à deux, fidèle à ce qui se passe réellement dans un machine virtuelle *Java Card*, permet des résultats consistants quand on manipule directement les bits.

Pour faire face aux dépassements de capacité et se conformer aux spécifications de Sun, on calcule d'abord les résultat exact pour chaque opération et on tronque le résultat au nombre de bits spécifié par `Z_bits`. En conséquence, le résultat final peut ne pas avoir le même signe que le résultat exact, mais c'est ce qui est requis par les spécifications.

Enfin, pour intégrer notre représentation des entiers à la formalisation de la machine virtuelle et en particulier à la représentation `valu` des valeurs, on fournit une conversion bi-directionnelle entre `Z_bits` et `Z`.

4.1.2 Programme *Java Card*

Les programmes *Java Card*, sont représentés en COQ sous une forme déjà liée, c'est-à-dire sans *constant pool*, sans chaînes de caractères et avec les index déjà résolus.

Ces données représentant un programme se résument alors dans trois structures : les interfaces, les classes et les méthodes. Pour chacune de ces structures, le type `Record` de COQ (en fait un type inductif à un seul constructeur avec pour arguments les champs définis dans `Record`) a été choisi. Il peut se rapprocher des concepts de `struct` en C et permet d'accéder facilement à chacun des champs qu'il contient.

Un programme *Java Card* n'est finalement qu'une structure constituée des listes des interfaces, des classes et des méthodes du programmes :

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

Les classes Une classe décrit un ensemble de méthodes et de variables, elle étend le plus souvent une autre classe, appelée sa *super-classe* et dont elle *hérite*, et éventuellement *implémente* des interfaces.

Dans le langage *Java Card*, pour des raisons de sécurité, chaque classe appartient à un *package* décrivant l'organisme créateur de cette classe et englobant éventuellement d'autres classes. Cela permet des communications privilégiés entre classes appartenant à un même *package*.

Toutes ces informations, listes d'héritage, liste de méthodes de la classe, liste des méthodes héritées redéfinies et liste des méthodes d'interface implémentées, liste des types des variables de classe et finalement le *package* auquel appartient la classe sont décrites dans la structure COQ `Class` :

```
Record Class : Set := {
  (* the list of all the super classes of this class *)
  super           : (list cap_class_idx);
  (* list of the virtual methods *)
  virtual_methods : (list Method);
  (* list of the package methods *)
  package_methods : (list Method);
  (* list of implemented interfaces and implemented methods *)
```

```
  int_methods     : (list cap_interf_idx *
                             (list (bool*class_vmethod_idx)));
  (* list of types of class variables *)
  class_var       : (list type);
  (* indentification of the owner package *)
  package         : Package;
  (* Index of class *)
  class_id        : cap_class_idx
}.
```

Les interfaces Une interface définit un ensemble de variables de méthode dites *abstraites* décrites uniquement par leur signature. Une classe implémente une interface si elle implémente toutes les méthodes abstraites de cette interface et si elle en reprend les variables. Une interface peut éventuellement étendre d'autres interfaces, dont elle *hérite*, en en reprenant toutes leur méthodes et variables.

Ces informations sur ces variables d'interface et les méthodes abstraites n'ont plus d'utilité après la liaison d'un programme. Elles sont directement intégrées aux classes implémentant des interfaces. Le *linker* vérifiera simplement à partir des ces informations qu'un classe implémente correctement toutes les méthodes décrites dans l'interface.

Ainsi la seule information encore utile après la liaison du programme est la liste d'héritage de l'interface (pour vérifier par exemple la compatibilité de deux tableaux d'interfaces) :

```
Record Interface : Set := {
  (* All the super interfaces of the interface. *)
  super_int : (list cap_interf_idx);
  (* index of the interface in the cap file *)
  Int_id : cap_interf_idx
  }.
```

Les méthodes Une méthode n'est qu'en fait qu'une liste d'instructions à exécuter appartenant à une classe. On retrouve donc dans la structure décrivant une méthode cette liste d'instruction et un pointeur vers la classe où elle a été définie (sous forme d'index dans la liste de classes du programme).

Néanmoins pour un meilleur contrôle lors de l'exécution d'une méthode, d'autres renseignements sont nécessaires, comme par exemple sa signature (pour un vérification des arguments passées), son nombre de variables locales (pour l'initialisation de son contexte d'exécution) ou sa liste de gestionnaires d'exceptions.

La structure décrivant une méthode est ainsi donnée par :

```
Record Method : Set := {
  (* Signature of the method, pair of domain / codomain *)
  signature    : ((list type)*type);
```

```
  (* Number of local variables. *)
  local        : nat;
  (* All the instructions to be executed. *)
  bytecode     : (list Instruction);
  (* Indicates whether a method is static or not *)
  statusstatic : bool;
  (* List of exception handlers *)
  handler_list : (list handler_type);
  (* index in the methods list of the cap file *)
  method_id    : cap_method_idx;
  (* owner Class index *)
  owner        : cap_class_idx
}.
```

4.1.3 Mémoire

La machine virtuelle *Java Card* requiert de l'espace mémoire pour l'exécution de programmes *Java Card*. Cette mémoire résume à un tas (*heap*) où sont stockés les objets crées par le programme, un tas statique (*static heap*) pour les variables d'instance statiques des objets et une pile d'environnement d'exécution de méthodes (*frame*).

Les objets Les objets crées pendant l'exécution d'un programme se limitent à des instances de classe et des tableaux. Ainsi, le type COQ décrivant un objet est-il naturellement défini par un type inductif à deux constructeurs, l'un de type un descriptif d'instance et l'autre un descriptif de tableau :

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

Pour les instances, les informations nécessaires sont, à l'image des types classes, stockées dans un enregistrement. Cet enregistrement a pour champs l'index de la classe dont l'objet considéré est une instance, les variables d'instance (comme une liste de termes de type `valu`), la référence au contexte auquel appartient l'instance et enfin deux booléens pour la gestion des points d'entrée (cf 2.4) :

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

Pour les tableaux, l'utilisation du type `valu` pour stocker leurs éléments conduirait à des informations redondantes. En effet, tous les éléments du tableau

devant avoir le même type, seule la valeur de l'élément est importante. Le type des éléments du tableau est alors renseigné par un des champs du descripteur de tableau.
Comme pour les instances de classe, les tableaux appartiennent à un contexte et disposent d'un accès privilégié, s'ils ont été déclaré comme globaux :

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

Tous les objets créés sont stockés dans une liste d'objets appelée tas (*heap*).

Les contextes d'exécutions Lors de l'exécution d'une méthode, un nouveau contexte (*frame*) est créé pour stocker les calculs effectués. Ce contexte, est rajouté à la liste des contextes déjà existants (*stack*), devient le contexte d'exécution courant, et est dépilé à la fin de l'exécution de la méthode.

Un contexte d'exécution de méthode contient principalement, une pile d'opérande, où sont dépilés les valeurs nécessaires à l'exécution d'une instruction et empilés les résultats. Il contient également un emplacement pour conserver les valeurs des variables locales de la méthode, sous forme de liste. Enfin, sont présents des références vers la méthode actuellement exécutée, vers la prochaine instruction à exécuter (pointeur d'instruction) et la référence au contexte auquel appartient la méthode (qui devient le contexte courant et joue un rôle primordial dans le mécanisme de *firewall*).

```
Record frame : Set := {
  opstack       : (list valu);      (* Operand stack *)
  locvars       : (list valu);      (* Local Variables *)
  method_loc    : cap_method_idx;   (* Location of the method *)
  context_ref   : Package;          (* Context Information *)
  p_count       : bytecode_idx      (* Program counter *)
}.
```

L'état de la machine virtuelle Un état de la machine virtuelle décrit l'ensemble de la mémoire à un instant donné. Il reprend donc les structures de *stack* et *heap* déjà décrites et y rajoute un tas statique, conservant sous forme de liste les valeurs des variables de classe déclarées statiques, dans une structure de triplet :

```
Definition jcvm_state := static_heap*heap*stack.
```

4.2 Instructions

La machine virtuelle supporte 110 instructions différentes. En raison de la forte similitude entre certaines de ces instructions dont le comportement ne diffère souvent que par le type des valeurs utilisées, ces 110 instructions ont pu entièrement être traitées par environ 50 fonctions COQ.

Dans la formalisation en COQ de la machine virtuelle, toutes les instructions ont une signature semblable, ne différant que par le nombre d'opérandes accepté :

$$\texttt{jcvm\_state} * \texttt{operands} \rightarrow \texttt{returned\_state}$$

où `returned_state` est un type inductif COQ à deux constructeurs traduisant le bon déroulement de l'exécution. Un premier constructeur indique que l'exécution s'est correctement passée et prend comme argument un terme de type `jcvm_state`, l'état de retour de l'instruction ; l'autre constructeur indique qu'une exception a été levée et qu'elle n'a pas pu être rattrapée, il prend comme argument l'exception levée et l'état de la machine virtuelle au moment où l'exception a été levée :

```
Inductive returned_state: Set :=
  Normal   :       jcvm_state->returned_state |
  Abnormal : xcpt->jcvm_state->returned_state.
```

Les instructions composant le bytecode de la machine virtuelle *Java Card* peuvent se regrouper en 8 catégories :

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

Pour la plupart des instructions, le mécanisme de l'exécution suit le même schéma :

- déstructuration de l'état la machine virtuelle pour en récupérer chacune des composantes et vérifier que la liste de *frames* n'est pas vide ;
- récupération d'informations dans les composantes de l'état de la machine virtuelle, comme par exemples d'éléments de la pile d'opérandes du contexte courant d'exécution ;
- construction d'un état de retour, nécessitant par exemple la mise à jour de la pile d'opérande.

4.3 Détails d'une instruction

Comme exemple de l'exécution d'une instruction, on s'intéresse au bytecode `invokevirtual` qui a pour but l'appel d'une méthode virtuelle. Lors de cet appel, les arguments de la méthode appelée sont pris et enlevés de la pile d'opérande du contexte d'exécution courant. Un nouveau contexte d'exécution pour la méthode appelée est créé et empilé à la pile de contextes. Enfin, les arguments sont placés dans la pile d'opérande de ce nouveau contexte.

La sémantique de cette instruction est prise en compte par la fonction COQ décrite ci-dessous :

```
Definition INVOKEVIRTUAL := [nargs:nat][nm:class_vmethod_idx]
[typmet:bool][state:jcvm_state][cap:jcprogram]
Cases state of
(sh, (hp, ((cons h lf) as s))) =>
  (* nargs must be greater than zero *)
  Cases nargs of
  O     => (ThrowCoqError args_error state) |
  (S _) =>

    (* Extraction of the object reference (the nargsth element) *)
    Cases (Nth_func (opstack h) nargs) of
    error => (ThrowCoqError opstack_error state) |
    (value (tx,vx)) =>

      (* Tests if this element is a reference *)
      Cases tx of
      (Ref _) =>

        (* tests if the reference is null *)
        Cases (test_NullPointer vx) of
        true  => (ThrowException NullPointer state cap) |
        false =>

          (* Extraction of the referenced object *)
          Cases (Nth_func hp (absolu vx)) of
          (value nhp) =>

            (* Tests if the object is an instance *)
            Cases nhp of
            (Instance ti) => (res_invokevirtual (reference ti) nargs
                               nm nhp typmet state cap) |

            (* In the case of an array, the associated class is Object *)
            (Array ta)    => (res_invokevirtual java_lang_Object nargs
                               nm nhp typmet state cap)
```

La fonction `new_frame_invokevirtual`, appelée par `res_invoke`, retourne un élément de type `jcvm_state` et, si nécessaire, met-à-jour la pile de contextes d'exécution.

```
Definition new_frame_invokevirtual :=
[nargs:nat][m:Method][nhp:obj][state:jcvm_state][cap:jcprogram]
Cases state of
(sh, (hp, ((cons h lf) as s))) =>
```

```
  (* Extraction of the list of arguments *)
  Cases (l_take nargs (opstack h)) of
  (value l) =>
    Cases (l_drop nargs (opstack h)) of
    (value l') =>

      (* security checking *)
      Cases (jcre_invokevirtual h nhp) of
      (value u) =>(ThrowException u state cap)|
      error     =>

        (* objectref is removed from the list of arguments *)
        Cases (drop_last l) of
        (value lis) =>

          (* then a signature check is performed *)
          Cases (signature_verification lis (Fst (signature m)) cap) of

          (* in case of success, the stack of frame is updated *)
          true => (Normal (sh, (hp,
                           (cons (Build_frame (nil valu)
                                              (make_locvars l (local m))
                                              (method_id m)
                                              (get_owner_context nhp)
                                              (0))
                                    (cons (Build_frame l'
                                                       (locvars h)
                                                       (method_loc h)
                                                       (context_ref h)
                                                       (p_count h))
                                     (tail s)))))) |
```

La machine virtuelle peut dans certaines conditions lever des exceptions. Par exemple :

- une exception `Nullpointer` est levée si l'objet extrait de la pile d'opérande est une référence nulle ;
- une exception de sécurité est levée si la politique de sécurité de *Java Card* est violée.

La fonction suivante, appelée par `new_frame_invokevirtual`, effectue les tests de sécurité à l'exécution nécessaires sur les objets (un objet est ou bien une instance de classe ou un tableau).

```
Definition jcre_invokevirtual := [h:frame][nhp:obj]
Cases nhp of
(* nhp is an instance*)
(Instance ti) =>

  (* tests if the active context is the owner context *)
  Cases (eqb_AID (AID_pi (context_ref h)) (AID_pi (owner_i ti))) of
  true  => Error |
  false =>
```

```
    (* tests if the the instance is an Entry Point *)
    Cases (eqb (ptE ti) is_ptE) of
    true  => Error |
    false =>

      (* tests if the active context is the JCRE context *)
      Cases (eqb_AID (AID_pi  (context_ref h)) jcre_AID) of
      true => Error |
      _    => (Value Security)
      end
    end
  end |

(* the object is an array. Tests are similar to the previous case *)
(Array ta) => ...
```

4.4 Exécution

Un programme *Java Card* se présente sous la forme d'un fichier `.cap`. Un outil nommé *JCVMTool* développé à l'INRIA par Bernard SERPETTE et écrit en *Java* traduit ce fichier vers un fichier COQ. Il effectue pour cela la liaison du programme et extrait du fichier `.cap` les informations nécessaires à notre formalisation pour représenter un programme *Java Card*.

L'utilisateur de la formalisation doit après cette étape de conversion, préciser le nombre de pas d'exécutions du programme qu'il souhaite effectuer. En effet, COQ n'acceptant que les programmes primitifs récursifs, donc terminant, il n'est pas possible de faire tourner de manière autonome un programme *Java Card* jusqu'à sa terminaison. Il faudrait d'abord prouver que le programme *Java Card* termine.

La fonction d'exécution de la machine virtuelle construit, à partir du fichier COQ compilé représentant le programme *Java Card*, l'état initial, en initialisant par exemple le tas statique ou en pré-allouant des instances d'objets, et exécute le nombre d'instructions précisé par l'utilisateur. Le résultat de retour de cette fonction est un terme de type `returned_state`, comme pour toutes les instructions.

De très nombreux tests ont été effectués sur la machine virtuelle. Malgré l'importance de la formalisation (7 500 lignes environ), ces tests n'ont dévoilés que très peu d'erreurs, principalement des erreurs d'index ou de mauvais argument. La grande rigidité de COQ vis-à-vis du traitement exhaustif des cas possibles est certainement à l'origine de ce comportement correct.

5 Une première abstraction : l'abstraction de types

La formalisation complète de la machine virtuelle *Java Card* est la première étape pour la réalisation d'abstractions. Parmi celles-ci, une des plus courantes est l'abstraction de types. Elle consiste à abstraire les valeurs, ici les valeurs de la machine virtuelle pour ne garder que leur type.

5.1 Modifications apportées à la machine virtuelle

La machine virtuelle abstraite est en quelque sorte une version allégée de la machine virtuelle précédemment réalisée. On redéfinit d'abord le type `valu` non plus comme `type*Z` mais simplement comme `type`.

Toutefois, des valeurs particulières, celles de type `Return_Address` sont nécessaires à l'exécution de programmes. Elles indiquent une position du pointeur d'instruction et sont utilisées par l'instruction `ret`. On est alors amené à modifier le système de types pour placer ces valeurs dans type `Return_Address` lui-même. Le constructeur `Return_Address` est donc modifié pour prendre en argument une valeur de pointeur d'instruction.

Les modifications sur `valu` entraînent de fait la suppression de tous les calculs dans les instructions.

Le code de l'instruction d'addition se résume ainsi à l'extraction de deux types dans la pile d'opérande du contexte d'exécution courant et à l'empilement du type du résultat, sans autres calculs.

De ce fait, la pile de contextes d'exécution peut se restreindre à un seul élément. En effet, la méthode n'effectuant plus de calculs, seul le type de retour de la méthode éventuellement appelée est utile. Il n'y a plus réellement d'appels de méthode.

Le tas même devient inutile, la valeur des objets créés, instances ou tableaux, importe peu. Seul compte le type des objets créés, toujours poussé dans la pile.

Ainsi, l'état de la machine virtuelle abstraite se résume alors à :

```
Definition abs_jcvm_state := static_heap*frame.
```

Le tas statique ne contient plus que le type des variables d'instance statiques, lequel type ne peut être retrouvé autrement.

Certaines instructions, par exemple les instructions de comparaison avec branchement (tels que `ifnull`), ont la particularité d'aboutir à deux points différents du programme suivant la valeur placée dans la pile d'opérande. En l'absence des informations sur les valeurs, la fonction d'exécution rend alors, sous forme liste, les différents états de retour possibles, avec des pointeurs d'instruction différents.

Enfin, dans le but de faire tourner les machines virtuelles abstraites et concrètes ensemble, les fonctions ou les variables sont renommées pour de ne pas surcharger leur nom et les distinguer clairement.

5.2 Détails d'une instruction abstraite

L'abstraction menée sur les types conduit, pour rester sur l'exemple de la section 4.3 sur la fonction `invoke_virtual`, à redéfinir sa formalisation en COQ sous la forme suivante :

```
Definition abs_INVOKEVIRTUAL := [nargs:nat][nm:class_vmethod_idx]
[typmet:bool][state:abs_jcvm_state][cap:jcprogram]
Cases state of
(sh, h) =>
  (* nargs must be greater than zero *)
  Cases nargs of
  O     => (abs_ThrowCoqError args_error state) |
  (S _) =>

    (* Extraction of the object reference (the nargsth element)*)
    Cases (Nth_func (topstack h) nargs) of
    (value x) =>
      Cases x of
      (tRef (tRef_instance c)) =>
        (abs_res_invokevirtual c nargs nm typmet state cap) |
      (tRef (tRef_array _))    =>
        (abs_res_invokevirtual java_lang_Object nargs nm typmet state cap) |
      _                        =>
        (abs_ThrowCoqError type_error state)
       end |
    error => (abs_ThrowCoqError opstack_error state)
    end
  end |
_ => (abs_ThrowCoqError state_error state)
end.
```

où l'on remarquera que les tests sur la nullité de la référence extraite et sur la recherche de cette référence dans le tas ont disparus. En effet, l'adresse dans le tas d'un objet est une valeur donc disparaît avec l'abstraction et le tas même est devenu inutile.

La fonction suivante appelée par `abs_res_invokevirtual` et construisant l'état de retour devient quant à elle :

```
Definition new_absframe_invokevirtual :=
[nargs:nat][m:Method][state:tjcvm_state][cap:jcprogram]
Cases state of
(sh, h) =>
  Cases (l_take nargs (abs_opstack h)) of
  (value l) =>
    Cases (l_drop nargs (abs_opstack h)) of
    (value l') =>
      Cases (drop_last l) of
      (value lis) =>
        Cases (signature_verification (valulist_to_typelist lis)
                                     (Fst (signature m)) cap) of
        true => (abs_Normal (sh, (Build_abs_frame (cons (Snd (signature n)) l')
                                     (abs_locvars h)
                                     (abs_method_loc h)
                                     (abs_context_ref h)
                                     (S (abs_p_count h))))) |
        false => (abs_ThrowCoqError signature_error state)
```

```
        end |
      _  =>  (abs_ThrowCoqError signature_error state)
      end |
    error => (abs_ThrowCoqError opstack_error state)
    end |
  error => (abs_ThrowCoqError opstack_error state)
  end|
_ => (abs_ThrowCoqError state_error state)
end.
```

Les tests de sécurité, reposant sur des propriétés dynamiques des objets, non plus de raison d'être réalisés ici.

Le type de retour, sans exécution de la méthode appelée, est placé sur la pile d'opérande de la méthode appelante.

5.3 Preuve de correction de l'abstraction

Pour assurer l'adéquation de la machine virtuelle abstraite par rapport la machine virtuelle concrète, on est amené à montrer la correction de l'abstraction réalisée.

On peut aisément, par simple projection, passer du système de types de la machine virtuelle concrète au système de types la machine virtuelle abstraite. On peut alors étendre cette fonction d'abstraction aux états de la machine virtuelle :

$$\alpha : jcvm\_state \rightarrow jcvm\_state^{\sharp}$$

La propriété de correction s'exprime par le diagramme de la figure 2. On s'intéresse à l'exécution simultanée des deux machines virtuelles et il convient alors de montrer que le diagramme est commutatif.

L'exécution abstraite d'une instruction pouvant mener à deux états de retour, il s'agit de vérifier pour la partie droite du diagramme non pas que le résultat de l'exécution abstraite est l'abstraction du résultat de l'exécution concrète, mais que les résultats possibles de l'exécution abstraite contient l'abstraction du résultat de l'exécution concrète.

D'autre part, les exceptions standard *Java Card* (telles que `NullPointer Exception`, `Arithmetic Exception`), éventuellement levées au moment de l'exécution par la machine virtuelle concrète, n'étant pas gérées par la machine virtuelle abstraite, il faut faire l'hypothèse que l'exécution concrète ne lève pas d'exceptions. Les seules exceptions susceptibles donc d'être levées par les deux machines virtuelles sont des exceptions dues à l'environnement d'exécution de la machine virtuelle, c'est-à-dire COQ lui-même, et qui sont nommées `CoqError` dans la formalisation.

Cette propriété s'exprime en COQ par le théorème suivant :

```
Theorem diag_commut : (cap:jcprogram)(state:jcvm_state)(i:Instruction)
  ((x:xcpt) (~x=CoqError->
              ~(exec_instr i state cap)=(Abnormal x state))->
```

$$\begin{tikzcd}
jcvm_state \arrow[r, "exec_instr(i)"] \arrow[d, "\alpha"] & returned_state \arrow[d, hook, "\alpha"] \\
jcvm_state^\sharp \arrow[r, "exec_instr^\sharp(i)"] & (list\ returned_state^\sharp)
\end{tikzcd}$$

FIG. 2 – *Diagramme commutatif des exécutions concrètes et abstraites*

```
(In (rs2absrs (exec_instr i state cap))
       (abs_exec_instr i (s2abss state) cap)).
```

et se démontre par cas sur les instructions. En raison du nombre d'instructions à traiter et du grand nombre de cas à traiter pour chaque instructions, ce théorème n'a pour l'instant été montré que pour quelques instructions seulement.

6 Application : un vérificateur de bytecode pour *Java Card*

La vérification de bytecode est une analyse statique de programmes permettant de s'assurer que des erreurs telles qu'un branchement d'instruction incorrect ou une pile ne contenant pas le bon nombre ou type d'arguments ne se produiront pas lors de l'exécution. Même si les compilateurs *Java*, utilisés pour les *Java Card*, ne produisent normalement que du code ne provoquant pas d'erreurs à l'exécution, il est possible de lancer un programme *Java* ou *Java Card* de source non sûre sous forme de *class file*. Il faut alors vérifier que ce programme dont on ne connaît le code source n'essaye pas d'exploiter la machine virtuelle.

6.1 Les fonctionnalités d'un vérificateur de bytecode

Le vérificateur de bytecode doit s'assurer qu'à tout point du programme, quel que soit le chemin suivit pour atteindre ce point, les propriétés suivantes sont vérifiées :

– la taille de la pile d'opérande est toujours la même et contient les mêmes types de valeurs ;
– aucun accès à une variable locale n'est fait sans en connaître son type ;
– les méthodes sont appelées avec les arguments appropriés ;
– les variables d'instance sont assignées avec des valeurs au type approprié ;
– les instructions ont les types appropriés sur la pile d'opérandes et dans les variables locales.

Cette étape de vérification améliore également les performances de la machine virtuelle. Les tests à effectuer à l'exécution pour vérifier les contraintes précédentes peuvent ainsi être éliminés, la machine virtuelle assumant que ces tests ont déjà eu lieu.

Dans notre modèle, toutes les contraintes de 6.1 sont aussi vérifiées lors de l'exécution d'un programme. Ainsi notre machine virtuelle s'assimile à une machine virtuelle *défensive*. Il est possible d'enlever ses contraintes sur la base que la vérification du bytecode a été achevée avec succès.

Par exemple dans l'exécution d'une instruction, les tests sur l'état de pile pour l'extraction d'une valeur conduisent à vérifier si cette pile est vide ou non :

```
Cases (head (opstack h)) of
  (value v) => ... |
  error => (ThrowCoqError opstack_error state)
```

Avec la preuve, fournie par la vérification réussie du bytecode, que la pile contient effectivement un élément, ce raisonnement par cas n'aurait plus lieu d'être.

La formalisation de la machine virtuelle où les tests défensifs seraient remplacés sur le preuve que ces tests ont déjà été passés avec le vérificateur de bytecode fournit une propriété intéressante lors de l'extraction vers *Objective CAML* par exemple. On obtiendrait alors d'une part le vérificateur de bytecode et d'autre part une machine virtuelle non défensive (les termes de preuves ayant été enlevés) !

La vérification de bytecode s'effectue après l'étape de liaison. C'est justement sous une forme déjà liée que les programmes *Java Card* sont formalisés en COQ. Ainsi, la formalisation est bien adaptée à ce processus de vérification.

6.2 Algorithme

Pour réaliser la vérification de bytecode, on teste chaque méthode du programme indépendamment. Pour chacune de ces méthodes, on exécute les instructions de la méthode considérée en ne gardant que les informations de types (comme en 5). Les tests effectués lors de l'exécution de la machine virtuelle abstraite garantissent que les propriétés 6.1, à l'exception de la première de ces propriétés, sont vérifiées. En effet, tous ces tests, imposés par l'analyse de cas dans COQ (par exemple sur les listes) ou bien rajoutés aux définitions des instructions, aboutissent à un état anormal, avec une exception de type `CoqError`, et ces inconsistances du programme sont repérées.

Pour la première des propriétés de 6.1 (à chaque instruction, la taille de la pile d'opérande est toujours la même et contient les mêmes types de valeurs), l'analyse est plus délicate. Cette propriété est imposée par la possibilité de passer plusieurs fois grâce à des boucles ou à des sauts d'instructions plusieurs fois par un même point du programme. Pour la vérifier, cela requiert de mémoriser l'état de la machine virtuelle après chaque exécution d'une instruction de la méthode considérée. Si une instruction est exécutée à nouveau, l'état de retour doit être *unifié* avec l'état précédemment sauvegardé pour cette instruction, selon les règles suivantes :

- les piles d'opérande des deux états doivent avoir la même taille ;
- les types des valeurs sur la pile doivent être identiques, à l'exception des types de références ;

– les types des valeurs des variables locales doivent aussi être identiques, à l'exception des types de références ;
– l'unification de types de références, le cas échéant, aboutit à la première super-classe commune des deux types (cette super-classe existe toujours, `Object` étant super-classe de tous les types de références).

Si l'une de ces règles échoue, la vérification échoue également.

Les règles d'unification sur les types sont données exhaustivement dans l'annexe A.

Pour débuter la vérification sur une méthode, on place dans les variables locales le nombre et le type d'arguments attendus par la méthode, la pile d'opérande est, quant à elle, vide. Les sauvegardes de l'état de chaque instruction sont mises à un état non initialisé.

On lance ensuite l'exécution sur la première instruction de la méthode et on laisse l'exécution se poursuivre instruction par instruction. Cependant, certaines instructions, comme les tests avec branchements, peuvent aboutir à deux points différents de la méthode, en fonction d'une valeur de la pile d'opérande. En l'absence de ces valeurs dans la pile d'opérande (on ne connaît que le type de ces valeurs), il faut poursuivre l'exécution après ces instructions particulières aux deux points possibles de branchement, et effectuer à ces deux points l'étape d'unification avec le résultat de l'instruction.

Il existe une dernière possibilité de branchement, et donc d'unification, avec les gestionnaires d'exceptions. Il faut, pour chaque gestionnaire d'exception, exécuter le bloc de récupération (bloc `catch`) avec pour état initial du bloc une pile d'opérande contenant simplement le type de l'exception récupérée. Une unification a lieu à la fin du bloc, au point de retour du gestionnaire d'exception.

Lorsqu'après une étape d'unification, l'état de sauvegarde n'a pas varié, l'exécution peut s'arrêter pour ce chemin d'exécution particulier suivi. En effet, si l'état de sauvegarde n'a pas varié après l'étape d'unification, cela signifie que :

– un chemin d'exécution a déjà atteint ce point du programme, sans quoi l'état de sauvegarde aurait gardé un état non initialisé et le résultat de l'unification aurait placé l'état de sauvegarde dans un état initialisé donc différent ;
– la poursuite de l'exécution à partir de cette instruction ayant donc déjà été effectuée avec le même état lors du suivi du chemin ayant aboutit à l'état de sauvegarde, elle n'a donc plus à l'être.

6.3 Preuve de terminaison

L'algorithme de vérification de bytecode requiert une récursion sur la fonction d'exécution d'instruction. Pour implémenter cet algorithme en COQ sous forme de fonction récursive, il faut montrer que cet récursion termine car COQ n'accepte que les programmes récursifs terminant.

Cette preuve de terminaison n'est pas triviale comme peut l'être les preuves pour les fonctions définies en COQ par récurrence structurelle sur un argument entier par exemple. Il faut en effet dans ce cas exhiber une quantité décroissante strictement à chaque étape de récursion et baser la preuve de terminaison sur la preuve que cette quantité décroît.

Treillis des types On va montrer qu'à chaque étape de récurrence, le résultat de l'unification est, selon un ordre à définir, plus petit que l'état sauvegardé. L'unification étant basée sur le système de types de la machine virtuelle, on définit d'abord un ordre sur ce système de types. Cet ordre est donné, de manière simplifiée, sur le treillis complet de la figure 3.

On ajoute au système de types de la machine virtuelle, les types $\perp$ dénotant un type non initialisé et $\top$ pour désigner une incompatibilité de types. On distingue ensuite clairement les types primitifs (Short, Int, Void, ...) des types des références (interface, instance de classe, tableau) qui ne sont pas comparables, et dont par conséquence le résultat de l'unification entre deux de ces types est $\top$.

Parmi les types de références, la classe *Java* `Object` est la super-classe commune. Enfin, on introduit une comparaison entre les instances de classe et les interfaces, l'unification d'une instance de classe et d'une interface pouvant être une interface si elle est implémentée par l'instance de classe considérée.

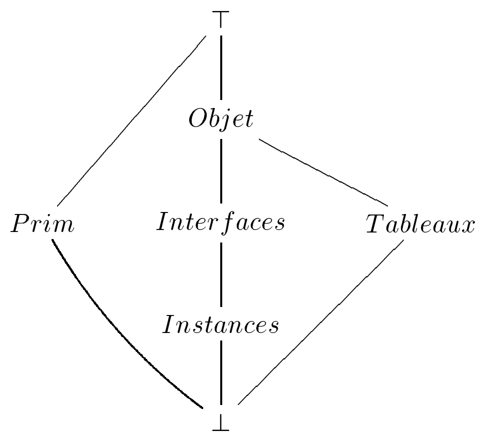

FIG. 3 – *Treillis des types de la machine virtuelle*

D'autres comparaisons ne sont pas indiquées sur le treillis, les instances de classes peuvent par exemple être comparées entre elles si l'une est une super-classe de l'autre, et de même pour les interfaces. Quant aux tableaux, on introduit une récursivité dans le treillis, le résultat de la comparaison de deux tableaux de types τ_1 et τ_2 étant le résultat de la comparaison de τ_1 et τ_2.

L'ordre $\preceq$ sur le treillis est donné par les règles d'inférence suivantes de la figure 4

Les interfaces (resp. classes) sont indicées de telle sorte que si I_m est une super-interface de I_n (resp. C_m est une super-classe de C_n) alors $m \leq_{\mathbb{N}} n$.

$$\Gamma \cup \{s : \texttt{type}\} \vdash \bot \preceq s \quad \text{(Bot)}$$

$$\Gamma \cup \{s : \texttt{type}\} \vdash s \preceq \top \quad \text{(Top)}$$

$$\Gamma \cup \{s : \texttt{type\_ref}\} \vdash s \preceq \mathit{Obj} \quad \text{(Obj)}$$

$$\Gamma \cup \{m,n : \mathbb{N}\} \vdash C(m) \preceq I(n) \quad \text{(ClI)}$$

$$\frac{\Gamma \cup \{m,n : \mathbb{N}\} \vdash m \leq_{\mathbb{N}} n}{\Gamma \cup \{m,n : \mathbb{N}\} \vdash I(m) \preceq I(n)} \quad \text{(Int)}$$

$$\frac{\Gamma \cup \{m,n : \mathbb{N}\} \vdash m \leq_{\mathbb{N}} n}{\Gamma \cup \{m,n : \mathbb{N}\} \vdash C(m) \preceq C(n)} \quad \text{(Cla)}$$

$$\frac{\Gamma \cup \{\sigma,\tau : \texttt{type}\} \vdash \sigma \preceq \tau}{\Gamma \cup \{\sigma,\tau : \texttt{type}\} \vdash A(\sigma) \preceq A(\tau)} \quad \text{(Arr)}$$

FIG. 4 – *Règles d'inférences de l'ordre sur le treillis des types*

Récursion bien fondée Pour montrer la terminaison de la fonction de vérification de bytecode, les arguments passés à cette fonction ne permettent pas d'utiliser une récurrence structurelle. Cependant, cette récurrence structurelle peut s'effectuer sur un argument supplémentaire, la preuve que l'ordre $\prec$ est bien fondé (suivant l'idée suivie dans [4]). En effet, en COQ, la bonne fondaison s'exprime par le prédicat d'accessibilité :

```
Inductive Acc [A:Set; R:A->A->Prop] : A->Prop := Acc_intro :
(x:A)((y:A)(R y x)->(Acc A R y))->(Acc A R x).
```

Ce prédicat caractérise les éléments x de A tels que toute chaîne décroissante à partir de x pour l'ordre R soit finie. Une relation bien fondée est une relation telle que tous les éléments de A soient accessibles :

```
Definition well_founded := [A:Set; R:(A->A->Prop)](a:A)(Acc A R a).
```

La récurrence structurelle s'effectuera alors sur le constructeur `Acc_intro` du prédicat d'accessibilité.

Pour appliquer cette technique à l'algorithme de vérification de bytecode, il faut donc finalement montrer en COQ que:

1. à chaque appel de récursion, effectué lorsque le résultat de l'unification r est différent du résultat sauvegardé s, ce résultat r est plus petit que s :

$$r \not\simeq s \Rightarrow \phi_r(s) \prec s;$$

2. l'ordre $\prec$ est bien fondé pour les éléments du système de types de la machine virtuelle.

6.4 Bonne fondaison de l'ordre sur le treillis

Tel que décrit par les règles de la figure 3, l'ordre $\prec$ n'est pas bien fondé. Il est en effet possible de construire une chaîne infinie décroissante pour cet ordre :

1. par l'axiome (Top), on a $\top \succeq A[\bot]$;
2. par la règle (Arr), on déduit $A[\top] \succeq A[A[\top]]$;
3. par transitivité, $\top \succeq A[\top] \succeq A[A[\top]]$
4. par applications successives de la règle (Arr), on arrive à :

$$\top \succ A[\top] \succ A[A[\top]] \succ A[A[A[\top]]] \succ ...$$

On peut toutefois, en modifiant la règle (Arr), obtenir un ordre bien fondé. On définit d'abord la taille $|\sigma|$ d'un type σ comme le nombre de fois où le constructeur `Array` apparaît dans ce type. On définit ensuite la restriction $\sigma_{|m}$ d'un type σ aux m derniers constructeurs `Array` du type σ, par exemple :

$$Obj_{|2} = Obj \qquad A(A(Prim))_{|2} = A(A(Prim))$$
$$A(A(A(A(Prim))))_{|2} = A(A(Prim))$$

On remarquera que :

$$|\sigma| \leq_{\mathbb{N}} m \Leftrightarrow \sigma_{|m} = \sigma \tag{1}$$

L'ordre $\preceq$ devient donc paramétré par un entier m et on change la règle (Arr) pour la règle :

$$\frac{\Gamma \cup \{\sigma,\tau : \texttt{type}\} \vdash \sigma \preceq_m \tau \qquad |\sigma| \leq_{\mathbb{N}} m \qquad |\tau| \leq_{\mathbb{N}} m}{\Gamma \cup \{\sigma,\tau : \texttt{type}\} \vdash A(\sigma) \preceq_m A(\tau)} \text{ (Arrm)}$$

Ainsi, il n'est plus possible d'obtenir une chaîne décroissante de tableaux de longueur supérieure à m.

Pour assurer la correction de $\preceq_m$ vis-à-vis de $\preceq$, il faut montrer que pour un programme donné, on peut borner la taille des types utilisés dans ce programme. Il suffit pour cela d'une simple analyse statique effectuant un parcours complet du programme à la recherche du plus grand type utilisé.

Soit P un programme et $\mathcal{T}(P)$ l'ensemble des types statiques présents dans P. Soit m tel que $\forall \sigma \in \mathcal{T}(P) \quad |\sigma| \leq_{\mathbb{N}} m$, on a alors grâce à (1) :

$$\forall \sigma,\tau \in \mathcal{T}(P) \quad \sigma \preceq_m \tau \Leftrightarrow \sigma \preceq \tau$$

On définit ensuite la fonction $\phi_m : \texttt{type} \rightarrow (\mathbb{N}^2)^m$ définie par :

$$\phi_m(\sigma) = (\phi_m^0(\sigma), \ldots, \phi_m^{m-1}(\sigma))$$

avec

$$\phi_m^i(\sigma) = \begin{cases} (0,0) & \text{si } |\sigma| < i \\ \phi(\pi_i(\sigma_{|m|}) & \text{sinon} \end{cases}$$

où $\pi_i(\sigma)$ est le i-ième constructeur du type σ et enfin :

$$\phi(\perp) = (0{,}0) \quad ; \quad \phi(Prim) = (1{,}0) \quad ;$$
$$\phi(I(n)) = (2{,}n) \quad ; \quad \phi(C(n)) = (3{,}n) \quad ; \quad \phi(A[s]) = (4{,}0) \quad ;$$
$$\phi(Obj) = (5{,}0) \quad ; \quad \phi(\top) = (6{,}0)$$

Cela donne par exemple :

$$\phi_2(\top) = ((6{,}0),(0{,}0)) \quad ; \quad \phi_2(A[Prim]) = ((4{,}0),(1{,}0)) \quad ;$$
$$\phi_2(A[A[I(n)]]) = ((4{,}0),(2{,}n))$$

Pour montrer la bonne fondaison de $\prec_m$, on peut se ramener à la bonne fondaison d'ordres connus en utilisant la propriété suivante.

Soient $(A, \prec_A)$ et $(B, \prec_B)$ deux ensembles ordonnés et $f : A \rightarrow B$ monotone, c'est-à-dire telle que :

$$\forall x{,}y \in A : x \prec_A y \Rightarrow f(x) \prec_B f(y),$$

alors si l'ordre $\prec_A$ est bien fondé sur A, $\prec_B$ est bien fondé sur B.

On remarque alors que la fonction ϕ_m est monotone de (`type`, $\preceq_m$) dans $((\mathbb{N}^2)^m, \leq_{(\mathbb{N}^2)^m})$ avec $\leq_{(\mathbb{N}^2)^m}$ défini de manière lexicographique gauche-droite. Or on sait que si un ordre est bien fondé alors son extension lexicographique l'est aussi. $(\mathbb{N}, \leq_{\mathbb{N}})$ est bien fondé donc $((\mathbb{N}^2)^m, \leq_{(\mathbb{N}^2)^m})$ est bien fondé et par la propriété précédente, (`type`, $\preceq_m$) l'est donc également.

On étend ensuite l'ordre $\preceq$ sur les types par un ordre lexicographique sur listes de types que sont le tas statique, la pile d'opérande, les variables locales en remarquant que :

- les longueurs de ces listes sont connues ;
- elles sont alors isomorphes à des n-uplets ;
- si un ordre est bien fondé alors son extension lexicograhique l'est également.

Finalement on construit un ordre sur les états de retour de la machine virtuelle abstraite et on l'étend encore à une liste de ces états (la liste des états de sauvegarde).

6.5 Implémentation

On donne le nom `lt_lrs` dans COQ à l'ordre sur la liste d'états de retour et on prouve la propriété suivante :

```
Lemma decrease_upd_lrs : (lrs:(list abs_returned_state))
(rs,rs':abs_returned_state)(cap:jcprogram)
  (Acc (list abs_returned_state) lt_lrs lrs)
    ->~rs=(unify_rs rs rs' cap)
     ->(Acc (list abs_returned_state) lt_lrs (update_lrs lrs rs rs' cap)).
```

qui indique que si une liste d'états de retour est accessible et si l'unification avec un état de retour modifie cette liste alors la liste modifiée est aussi accessible.

La preuve ce cette propriété utilise le fait qu'à chaque étape d'unification, si le résultat r de l'unification de deux types s et t est différent de s alors $r \prec s$. Il s'agissait de la première des deux propriétés de 6.3 à montrer.

Il est possible de demander à COQ le terme de preuve correspondant au lemme `decrease_upd_lrs` et l'on remarquera qu'il est bien effectué une récurrence structurelle sur la preuve du prédicat d'accessibilité :

```
decrease_upd_lrs =
[lrs:(list treturned_state);
 rs,rs':treturned_state;
 cap:jcprogram;
 h:(Acc (list treturned_state) lt_lrs lrs);
 diseq:(~rs=(unify_rs rs rs' cap))]
    ...
      <[l:(list treturned_state)]
        ~rs=(unify_rs rs rs' cap)
        ->(Acc (list treturned_state) lt_lrs (update_lrs l rs rs' cap))>
        Cases h of
          (Acc_intro x H0) =>
           [H1:(~rs=(unify_rs rs rs' cap))]
            (H0 (update_lrs x rs rs' cap) (unif_lrs_decr x H1))
        end (unif_lrs_decr lrs diseq)) diseq).
```

L'hypothèse d'accessibilité `h` est décomposée en `(Acc_intro x H0)` et la récurrence se poursuit sur `H0` structurellement plus petit que `h`.

Il est enfin possible de prouver la terminaison du vérificateur de bytecode par le théorème suivant :

```
Theorem run : (lrs:(list abs_returned_state))(rs:abs_returned_state)
(m:Method)(cap:jcprogram)(Acc (list abs_returned_state) lt_lrs lrs)
  ->(list abs_returned_state).
Fix 5.
Intros.
Refine
(* Decomposes the returned state *)
Cases rs of
  (abs_Abnormal x s) => (l_update_nth lrs (get_abs_p_count rs) rs) |
  (abs_Normal s) =>
    (* Decomposes the saved returned state *)
    Cases (Nth_elt lrs (get_abs_p_count rs)) of
      error => lrs |
      (value (abs_Abnormal x s')) => lrs |
      (value (abs_Normal s')) =>
        (* Checks for equality with unification *)
        Cases (iseq_rs (abs_Normal s') (unify_rs (abs_Normal s') rs cap)) of
          (left _) => lrs |
          (right _) =>
            (* Grabs the next instruction *)
            Cases (Nth_elt (bytecode m) (get_tp_count rs)) of
```

```
            error => lrs |
            (value i) =>
              (* Executes again *)
              Cases (abs_exec_instr i s cap) of
                (cons r l') =>
                  (run (update_lrs lrs (abs_Normal s') rs cap) r m cap ?) |
                nil => lrs
              end
            end
        end
    end
end.
Apply decrease_upd_lrs ; Auto.
Qed.
```

La démonstration du théorème s'effectue, suivant la stratégie adoptée, par récurrence structurelle sur la preuve de l'accessibilité de la liste d'états de retour (`Fix 5`, le cinquième argument). On utilise la tactique `Refine` pour donner une preuve exacte du théorème (cette preuve étant l'algorithme du vérificateur de bytecode), avec un trou `?`. Ce trou correspond à l'accessibilité de l'unification de la liste d'états de retour, dont la démonstration nous est fournie par le lemme `decrease_upd_lrs`.

Pour l'utilisation de ce théorème, il faut montrer l'hypothèse d'accessibilité sur la liste `lrs` d'états de retour. Ceci nous est donné par la preuve que l'ordre sur la liste d'états de retour est bien fondé et qu'en conséquence toute liste d'états de retour est accessible.

L'ensemble des travaux effectuées en COQ pour montrer la terminaison du vérificateur de bytecode représente près de 3 000 lignes de preuves.

7 Conclusion

7.1 Travaux connexes

Le besoin de méthodes formelles dans le domaine des *smart cards* a pour origines la maîtrise de la complexité des nouveaux systèmes d'exploitation, et la certification à un niveau élevé d'une partie d'une *smart cards*. Beaucoup de travaux ont déjà été réalisés dans le domaines des *smart cards* mais aucun n'a achevé une preuve complète d'un programme ou d'une composante de la machine virtuelle.

Au niveau de la plate-forme Il y a eu de nombreux efforts pour formaliser les composantes d'une *smart card*. Le plus important travail à ce jour est celui de l'équipe Bali à Munich (voir par exemple [9]) qui a formalisé en Isabelle/HOL une large partie de la plate-forme *Java* dont :

– le système de types et la sémantique opérationnelle du langage source avec une preuve de la sûreté du typage ;

– le compilateur et un vérificateur de bytecode, avec la preuve de leur correction

Il y a eu aussi des efforts similaires par Syme [13] qui a formalisé la sémantique opérationnelle de *Java* en DECLARE, par Lanet et Requet [7] qui ont formalisé la majeur partie de la JCVM en B, et par Cohen [2] qui a formalisé une machine virtuelle *Java* en ACL2.

Au niveau applicatif Le projet LOOP [6] a développé un outil pour compiler des classes *Java* vers des théories PVS ou Isabelle. Janet et al [10] ont quant à eux utilisé la méthode B pour prouver la correction d'applets *Java Card*, comme par exemple un porte-monnaie électronique.

D'autres projets, comme Java Path Finder [5] ou Bandera [3], privilégient l'utilisation de techniques algorithmiques pour vérifier les propriétés de programmes *Java*. Par exemple, l'ensemble d'outils de Bandera combine différentes techniques d'analyses ou de transformations de programmes pour extraire du code source *Java* un modèle compact à états finis soumis à un *model-checker*.

7.2 Développements futurs

La formalisation actuelle de la machine virtuelle *Java Card* et de son environnement d'exécution est le point de départ de nombreux autres développements.

Si pour l'instant la machine virtuelle supporte tout le jeu d'instructions du langage, il reste à implémenter les API de *Java Card* pour pouvoir exécuter un programme utilisant ces API. Des recherches ont déjà été effectuées en ce sens, notamment sur la manière d'implémenter les méthodes dites natives (c'est-à-dire les méthodes n'étant pas associées à du code *Java Card* mais exécutées par le langage natif dans lequel a été écrite la machine virtuelle), néanmoins ces dernières restent nombreuses, plus d'une centaine, et certaines très compliquées (notamment celles liées à la gestion des APDU).

Il serait également intéressant de réécrire la machine virtuelle avec la preuve que la vérification du bytecode a réussi, suivant l'idée développée dans 6.1.

Dans l'optique de formaliser l'ensemble du langage *Java Card* en COQ, il est nécessaire de traiter le *bytecode verifier* (travaux décrits dans ce document), le *cap converter* (celui de Sun est pour l'instant utilisé avant le traitement du *JCVMTool*, écrit lui-aussi en *Java*), et le *compiler*. Cela conduira à une formalisation complète du langage.

L'étape de spécification et de vérification des applets conduira à l'utilisation de techniques suivantes :

– Interprétation abstraite, afin d'éliminer parmi des traces d'exécutions d'applets imposantes les informations inutiles ;
– Model checking, pour vérifier sur des programmes des propriétés exprimées par des formules de logiques temporelles.

Ce stage a donné lieu à de nombreux développements en COQ, aussi bien au niveau formalisation que démonstration. Il a aboutit à une formalisation inédite de la machine virtuelle dans son ensemble (avec la gestion des exceptions, des changements de contexte, ...), sous une forme, de plus, exécutable. Des travaux tels que le vérificateur de bytecode ont déjà pu être effectués sur la formalisation et montrent ainsi les capacités et les atouts apportés par cette formalisation.

Références

[1] 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, C. Murthy, C. Parent-Vigouroux, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant User's Guide. Version 6.3.1*, December 1999.

[2] R. M. Cohen. Defensive Java Virtual Machine Specification Version 0.5. Manuscript, 1997.

[3] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of ICSE'2000*, 2000.

[4] E. Giménez. A tutorial on recursive types in Coq. Technical Report RT-0221, INRIA, 1998.

[5] K. Havelund. Java PathFinder—A Translator from Java to Promela. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Proceedings of 5th and 6th International SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science*, page 152. Springer-Verlag, 1999.

[6] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes. *ACM SIGPLAN Notices*, 33(10):329–340, October 1998.

[7] J.-L. Lanet and A. Requet. Formal proof of smart card applets correctness. In J.-J. Quisquater and B. Schneier, editors, *Third Smart Card Research and Advanced Application Conference*, Louvain-la-Neuve, Belgium, September 1998.

[8] Gary McGraw and Edward Felten. *Java security.* SIGS Books and Multimedia, New York, USA, 1996.

[9] T. Nipkow, D. von Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proceedings of the International Summer School Marktoberdorf 1999.* IOS Press, 2000. To appear.

[10] Virginie Wiels Guy Zanon Pierre Girard Pierre Bieber, Jacques Cazin and Jean-Louis Lanet. Electronic purse applet certification: extended abstract. In Steve Schneider and Peter Ryan, editors, *Electronic Notes in Theoretical Computer Science*, volume 32. Elsevier Science Publishers, 2000.

[11] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.1.1 Runtime Environment (JCRE) Specification*, 2000.

[12] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.1.1 Virtual Machine (JCVM) Specification*, 2000.

[13] D. Syme. Proving Java type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 83–118. Springer-Verlag, 1999.

A Opérateur d'unification

L'opérateur d'unification $\sqcap$, non symétrique, est défini pour les types primitifs par :

$\nearrow$	$\perp$	Void	Boolean	Byte	Short	Int	Ret_n	Ref	$\top$
$\perp$	$\perp$	Void	Boolean	Byte	Short	Int	Ret_n	Ref	$\top$
Void	Void	Void	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
Boolean	Boolean	$\top$	Boolean	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
Byte	Byte	$\top$	$\top$	Byte	Short	Int	$\top$	$\top$	$\top$
Short	Short	$\top$	$\top$	Byte	Short	Int	$\top$	$\top$	$\top$
Int	Int	$\top$	$\top$	Byte	Short	Int	$\top$	$\top$	$\top$
Ret_m	Ret_m	$\top$	$\top$	$\top$	$\top$	$\top$	$Ret_m \sqcap Ret_n$	$\top$	$\top$
Ref	Ref	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	Ref	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

pour les types de références par :

$\nearrow$	$\perp$	$Prim$	Obj	C_2	I_2	A_σ	$\top$
$\perp$	$\perp$	$Prim$	Obj	C_2	I_2	A_σ	$\top$
$Prim$	$Prim$	$Prim$	$\top$	$\top$	$\top$	$\top$	$\top$
Obj	Obj	$\top$	Obj	Obj	Obj	Obj	$\top$
C_1	C_1	$\top$	Obj	$C_1 \sqcap C_2$	$C_1 \sqcap I_2$	Obj	$\top$
I_1	I_1	$\top$	Obj	Obj	I $\sqcap$ I'	Obj	$\top$
A_τ	A_τ	$\top$	Obj	Obj	Obj	$A_{\tau \sqcap \sigma}$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

avec les cas particuliers :

$$C_1 \sqcap C_2 = \begin{cases} C_1 & \text{si } C_1 \text{ est une super-classe de } C_2 \\ C_2 & \text{si } C_2 \text{ est une super-classe de } C_1 \\ \text{Obj} & \text{sinon} \end{cases}$$

$$I_1 \sqcap I_2 = \begin{cases} I_1 & \text{si } I_1 \text{ est une super-interface de } I_2 \\ I_2 & \text{si } I_2 \text{ est une super-interface de } I_1 \\ \text{Obj} & \text{sinon} \end{cases}$$

$$C_1 \sqcap I_2 = \begin{cases} I_2 & \text{si } C_1 \text{ implémente } I_2 \\ \text{Obj} & \text{sinon} \end{cases}$$