# Un Cadre Sémantique pour la Correction de la Tokenisation de Java Card

Ewen Denney

Projet Lande
IRISA, Rennes

8 juillet 1999

# Cadre Général

**Compilateur**

Java (Card) class source code $\mapsto$ class file

**JVM**

package (class files) $\mapsto$ comportement

**Converteur**

package (class files) $\mapsto$ CAP file

**JCVM**

CAP file $\mapsto$ comportement

Nous voudrions savoir que cette transformation est *correcte*.

**Le Problème**

Comment formuler et prouver la correction, étant donné que les états des machines sont différents?

**Idée**

- Formuler une sémantique operationelle indépendante du format sous-jacent
- Construire une relation entre les deux formats qui est *observablement* l'identité.

## Deux Étapes

1. Donner une caractérisation abstraite de transformations possibles et démontrer que les propriétés abstraites garantissent la correction.

2. Démontrer qu'une telle transformation existe.

## Structure de la Preuve

1. Formuler une sémantique operationelle indépendante du format sous-jacent.
2. Abstraire les différences dans fonctions auxiliaires.
3. Interpréter les fonctions auxiliaires dans les deux formats.
4. Définir des ‘relations de couplage’ entre éléments correspondants dans les deux formats.
5. Prouver l’indépendance de la représentation des fonctions auxiliaires.

# Types Abstraits

## Types Basiques

$$
\begin{array}{rcl}
\texttt{Word} & ::= & \texttt{Object\_ref} \mid \texttt{Null} \mid \texttt{Boolean} \mid \texttt{Byte} \mid \texttt{Short} \\
\texttt{Math} & ::= & \texttt{Bool} \mid \texttt{Nat} \\
\gamma & ::= & \texttt{Word} \mid \texttt{Math}
\end{array}
$$

## Items

$$
\begin{array}{l}
\iota ::= \texttt{Package\_ref} \mid \texttt{Class\_ref} \mid \texttt{Static\_field\_ref} \\
\quad \mid \texttt{Instance\_field\_ref} \mid \texttt{Static\_method\_ref} \\
\quad \mid \texttt{Interface\_method\_ref} \mid \texttt{Virtual\_method\_ref}
\end{array}
$$

## Composants

$$
\kappa ::= \texttt{CP\_info} \mid \texttt{Method\_info} \mid \texttt{Classes} \mid \texttt{Methods} \mid \cdots
$$

## Types

$$\theta ::= \gamma \mid \iota \mid \kappa \mid \theta \times \theta' \mid \theta \to \theta'$$

## Types Définis

$$\begin{array}{rcl}
\texttt{Environment} & = & \texttt{Package_ref} \to \texttt{Package} \\
\texttt{Heap} & = & \texttt{Object_ref} \to \texttt{Obj} \\
\texttt{Constant_pool} & = & \texttt{CP_index} \to \texttt{CP_info} \\
\texttt{Environment} & = & \texttt{Package_ref} \to \texttt{Package} \\
\texttt{Instruction} & = & \texttt{Invokevirtual CP_index} \mid \texttt{Checkcast Tcode} \ldots \\
\texttt{Bytecode} & = & \texttt{Instruction} + \texttt{Bytecode} \times \texttt{Bytecode} + \texttt{Config}
\end{array}$$

## Types Observables

$$o ::= \gamma \mid o \times o' \mid o \to o'$$

# Fonctions Auxiliaires

$$\texttt{constant\_pool} : \texttt{Class\_ref} \rightarrow \texttt{Constant\_pool}$$

$$\texttt{method\_nargs} : \texttt{Method\_ref} \rightarrow \texttt{Nat}$$

$$\texttt{super} : \texttt{Class\_ref} \rightarrow \texttt{Class\_ref}$$

$$\texttt{lookup} : \texttt{Class\_ref} \times \texttt{Method\_ref} \rightarrow \texttt{Class\_ref} \times \texttt{Bytecode}$$

$$\texttt{static\_val} : \texttt{Field\_ref} \rightarrow \texttt{Value}$$

*etc.*

# invokevirtual

$$\frac{\begin{array}{l} stat\_mref := \texttt{constant\_pool}\ (c)(i) \\ n := \texttt{method\_nargs}(stat\_mref) \\ \langle dyn\_cref, \_\rangle := heap(r) \\ \langle m\_class, m\_code\rangle := \texttt{lookup}(dyn\_cref, stat\_mref) \end{array}}{\begin{array}{l} \langle \texttt{invokevirtual}\ i, a_1 \ldots a_n :: r :: s, l, c\rangle \\ \quad \Rightarrow \langle\langle m\_code, \langle\rangle, a_1 \ldots a_n :: r, m\_class\rangle, s, l, c\rangle \end{array}}$$

# Interprétations

Pour chaque type, $\theta$, nous donnons deux interprétations:

$$[\![\theta]\!]_{JVM} \quad \text{et} \quad [\![\theta]\!]_{JCVM}$$

et une relation

$$R_\theta : [\![\theta]\!]_{JVM} \longleftrightarrow [\![\theta]\!]_{JCVM}$$

Pour chaque fonction auxiliaire, $f : \theta \to \theta'$, nous donnons deux interprétations

$$[\![f]\!]_{JVM} \quad \text{et} \quad [\![f]\!]_{JCVM}$$

tel que

$$[\![f]\!]_{JVM} \; R_{\theta \to \theta'} \; [\![f]\!]_{JCVM}$$

# lookup (JVM)

```
lookup_name (dyn_class, (sig, stat_class)) =
              let
               stat_pk = pack_name(stat_class)
               dyn_pk  = pack_name(dyn_class)
               (_,_,_,_,meth_stat,_,_)
                      = env_nom (stat_pk) (stat_class)
               (_,super,_,_,_,meth_dyn,_,_)
                      = env_nom (dyn_pk) (dyn_class)
               (stat_flags,_,_,_,_,_,_) = meth_stat(sig)
               in
               if meth_dyn(sig) = undefined
               then lookup_name(super, (sig, stat_class))
               else if
                    stat_flags(protected) or stat_flags(public)
                    or dyn_pk = stat_pk
                    then let (_,_,_,_,_,code,_) = meth_dyn(sig)
                            in (dyn_class,code)
                    else  lookup_name(super, (sig, stat_class))
```

# lookup (JCVM)

```
lookup_tok (dyn_class_ref, (tag, stat_class_ref, method_tok)) =

case tag of

CONSTANT_virtual_method_ref, CONSTANT_super_method_ref ->


let methods = method_array (dyn_class_ref)
in
let (..., super, public_base, public_table, package_base, package_table,...)
                                                       : Class_info =
    class_info(dyn_class_ref)
in
if method_tok div 128 = 0 then /* public */
   if method_tok >= public_base then
      let method_offset = public_table[method_tok - public_base]
        in
        if method_offset <>  0xFFFF
        then (dyn_class_ref, methods[method_offset].Bytecode)
        else /* look in superclass */
           lookup_tok(super, (tag, stat_class_ref, method_tok))
   else /* look in superclass */
```

```
      lookup_tok(super, (tag, stat_class_ref, method_tok))
else /* package */
   if method_tok >= package_base /\
      (same_package(stat_class_ref, dyn_class_ref)
          \/ tag = CONSTANT_super_method_ref)
   then
      let method_offset = package_table[method_tok mod 128 - package_base]
        in (dyn_class_ref, methods[method_offset].Bytecode)
   else /* look in superclass */
      lookup_tok(super, (tag, stat_class_ref, method_tok))
```

## $R_{\texttt{Virtual\_method\_ref}}$

- $\langle c\_name, sig\rangle \; R_{\texttt{Virtual\_method\_ref}} \; \langle c\_ref, m\_tok\rangle \wedge$
  $\langle c'\_name, sig\rangle \; R_{\texttt{Virtual\_method\_ref}} \; \langle c'\_ref, m'\_tok\rangle \wedge$
  $c'\_name < c\_name \wedge$
  $(package(sig) \Rightarrow same\_package(c\_name, c'\_name))$
  $\Rightarrow m\_tok = m'\_tok$

- $new\_method\,(sig, c\_name) \;\wedge\; public(sig) \wedge$
  $(c\_name, sig) \; R_{\texttt{Virtual\_method\_ref}} \; (c\_ref, m\_tok) \Rightarrow$
  $\forall c'\_ref > c\_ref\,.\,\forall m'\_tok \in c'\_ref\,.\,m\_tok > m'\_tok$

- $\langle c\_name, sig\rangle \; R_{\texttt{Virtual\_method\_ref}} \; \langle c\_ref, m\_tok\rangle \Rightarrow$
  $(public(sig) \Rightarrow 0 \leq m\_tok \leq 127) \wedge$
  $(package(sig) \Rightarrow 128 \leq m\_tok \leq 255)$

# Relations Logiques

Une famille de relations qui respecte la structure des types. Pour chaque type, $\theta$, nous définissons une relation $R_\theta : S_\theta \leftrightarrow S'_\theta$ :

**Types simples**

$$R_T = \text{une relation}$$

**Produits**

$$(a, b)\ R_{\theta_1 \times \theta_2}\ (a', b') \iff a\ R_{\theta_1}\ a' \ \wedge\ b\ R_{\theta_2}\ b'$$

**Fonctions**

$$f\ R_{\theta_1 \to \theta_2}\ f' \iff \forall a, a' \,.\, a\ R_{\theta_1}\ a' \Rightarrow f a\ R_{\theta_2}\ f' a'$$

# lookup

$$\mathtt{lookup} : \mathtt{Class\_ref} \times \mathtt{Method\_ref} \rightarrow \mathtt{Class\_ref} \times \mathtt{Bytecode}$$

$$\begin{tikzcd}
{[\![\mathtt{Class_ref} \times \mathtt{Method_ref}]\!]_{JVM}} \arrow[r] \arrow[d, leftrightarrow, "R_{\mathtt{Class_ref} \times \mathtt{Method_ref}}"'] & {[\![\mathtt{Class_ref} \times \mathtt{Bytecode}]\!]_{JVM}} \arrow[d, leftrightarrow, "R_{\mathtt{Class_ref} \times \mathtt{Bytecode}}"] \\
{[\![\mathtt{Class_ref} \times \mathtt{Method_ref}]\!]_{JCVM}} \arrow[r] & {[\![\mathtt{Class_ref} \times \mathtt{Bytecode}]\!]_{JCVM}}
\end{tikzcd}$$

# Théorèmes

1. Si $R_\gamma = id_\gamma$ pour tout type basique, $\gamma$, alors $R_\theta = id_\theta$ pour tout type observable $\theta$.

2. Si pour toute fonction auxiliaire $f : \theta \to \theta'$, $[\![f]\!]_{JVM}\ R_{\theta\to\theta'}\ [\![f]\!]_{JCVM}$, alors pour tout bytecode *code*, $[\![code]\!]_{JVM}\ R_{\mathtt{State}\to\mathtt{Bytecode}\times\mathtt{State}}\ [\![code]\!]_{JCVM}$.

# Bilan et Perspectives

On a prouvé que si une famille $\{R_\theta\}_{\theta \in Type}$ satisfait certain propriétés, alors elle est 'correcte'.

- Il reste à prover qu'une telle famille existe (extraction, raffinement).
- Plusieurs lacunes – interfaces, exceptions, spécialisation des instructions ...
- Types dépendents pour un modèle plus élégant et précis.