Machines Virtuelles Applicatives

Génération de bytecode implémentant les streams en JavaScript

Fonctionnement de notre solution

Notre solution s'articule en trois étapes, dont deux seulement ont demandé à être remaniées depuis le lab 3.

1. Génération de méthodes « built-in »

Au lancement de la VM, les méthodes de création « built-in » sont chargées (au même niveau que la méthode « print » vue dans les labs). Ces méthodes placent dans l'environnement des JSObjects contenant des méthodes de gestion des streams. Par simplicité, on dit que ces JSObjects sont de type « stream ».

Les streams pour la vm sont des JSObjects disposant des méthodes next et hasnext.
Un JSObject de type stream contient les valeurs et les sort avec ses méthodes next et hasnext lorsqu'un forEach (ou autre consumer) est détecté côté JS.

2. Interprétation

Lorsqu'une opération est appelée côté JS, la JVM appelle l'objet MethodHandle du JSObject correspondant. Celui-ci, fait alors appel au bytecode généré à la

L'interprétation du bytecode, quant à elle, n'a pas été retouchée et est identique à celle du lab 3.

Méthode de travail et difficultés

Étant donné les emplois du temps chargés de chacun des membres du groupe, nous avons jugé plus utile de chercher chacun de notre côté des pistes et de mettre en commun nos avancées.
Il faut bien le dire, la génération du bytecode nous aura posé un gros problème. La difficulté venant de la complexité du code à générer et du débugging laborieux en cas d'exception levée.

Au final, Yann Sanquer a trouvé une solution astucieuse :

- coder l'opération voulue (generate, filter…)
- faire interpréter ce code par la JVM
- reprendre le bytecode généré, et l'optimiser

Des optimisations ont été apportées (au niveau des labels, principalement) à ce bytecode.

A partir de là, il était possible de créer toutes méthodes de l'API stream en reprenant la méthode « built-in » de « print » et le bytecode obtenu.