



M1 INFORMATIQUE - PROJET DE PC2R

RAPPORT

---

## Arènes Vectorielles Synchrones

---

*Étudiants :*  
Suxue LI  
Julien XAVIER

13 avril 2019

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Structure générale du projet</b>	<b>2</b>
2.1	Principe de communication . . . . .	2
2.2	server.ml . . . . .	2
2.3	main_direct.ml & main_trace.ml . . . . .	2
2.4	makefile & readme.md . . . . .	2
<b>3</b>	<b>Choix d'implémentation</b>	<b>3</b>
3.1	Valeurs . . . . .	3
3.2	Registres . . . . .	4
3.3	Instructions . . . . .	4
<b>4</b>	<b>Optimisation APPTERM : comparaison de la pile</b>	<b>4</b>
<b>5</b>	<b>Nouveaux jeux d'essai</b>	<b>5</b>
<b>6</b>	<b>Conclusion</b>	<b>5</b>

# 1 Introduction

Le devoir de programmation de cette année a pour objectif de réaliser une application clients-serveur permettant à des utilisateurs de jouer simultanément à un jeu de course de voitures dans l'espace représenté par une arène dite torique, signifiant une arène sous la forme d'un "donut".

Ainsi, le but du jeu est de permettre à plusieurs joueurs de s'affronter dans des sessions de jeu ayant alors pour objectif de passer à certains endroits particuliers de l'arène, appelés objectifs.

Afin de réaliser cette application, nous avons fait le choix d'implémenter la partie serveur en Ocaml et la partie client en Java.

Ce rapport présente d'abord la structure générale du projet, puis nos choix/difficultés d'implémentation et enfin un petit manuel de jeu.

## 2 Structure générale du projet

### 2.1 Principe de communication

Afin de réaliser la communication entre le serveur et les clients, nous avons utilisé un système de communication par socket, ce qui nous permet ici de réaliser des sessions TCP, afin de facilement et de manière fiable, envoyer des informations dans les deux sens de communication.

Celle-ci se fera via des protocoles afin de séparer et traiter chaque envoi/réception de manière individuelle.

### 2.2 server.ml

C'est ici que se passe tous les systèmes de communication et d'interprétation des protocoles côté Serveur.

On divise ici le travail en plusieurs parties :

- Une partie d'initialisation/threading (start), qui permet de créer toutes structures de données utilisées pour l'application du serveur et de démarrer les différents services en parallèles.
- Une partie réception (process), qui permet l'interprétation de chacun des services proposés par les protocoles de réception client->serveur.
- Une partie d'envoi (send), qui permet de préparer les commandes et d'envoyer aux différents clients les informations nécessaire au bon déroulement du jeu.

### 2.3 SpaceRun.java

Programme principal côté client. Ici, on a deux parties principales :

- La partie Interface graphique, faite à l'aide de JavaFX et de SceneBuilder, logiciel tierce permettant la création de scènes JavaFX et générant des fichiers .fxml. De plus, nous avons alors la génération d'éléments graphique (Drawer.java) et l'update des éléments de jeu (canvas) ou encore de description(descPane/listview) et le chat (chatbox).
- La partie réception, permettant lui aussi la réception et l'envoi de protocoles via la socket.

Ensuite, pour exécuter un programme bytecode sur la machine, nous avons écrit deux versions dans deux fichiers distincts : *main\_direct.ml*, et *main\_trace.ml*. Le premier permet d'obtenir simplement une sortie standard et une valeur de retour, tandis que le deuxième affiche l'état des registres de la mini-ZAM après chaque instruction évaluée, la trace de l'exécution "corrompt" l'affichage réel du programme, par contre la valeur de retour sera toujours affichée à la fin.

## 2.4 makefile & readme.md

Nous avons également écrit un *makefile* contenant plusieurs cibles, permettant de compiler les différents fichiers, pour créer des exécutables. Vous trouverez plus de précision concernant l'utilisation du makefile dans le fichier *readme.md* qui vous décrit plus en détail les différentes cibles.

## 3 Choix d'implémentation

### 3.1 Valeurs

La mini-ZAM manipule des valeurs d'un type *mlvalue* que nous avons choisi de définir ainsi :

Le constructeur Entier permet de construire des valeurs représentant les valeurs immédiates entières, mais également les booléens, et ().

Le constructeur Fermeture prend en argument un entier et une valeur de type *mlvalue* construite avec Env, pour représenter respectivement le pointeur de code ainsi que l'environnement d'exécution de la fermeture.

Le constructeur Env prend une liste de *mlvalue* constituant l'environnement à représenter. Env n'existait pas au début du projet, il a été introduit lors du développement. Nous n'avions pas de constructeur pour l'environnement, il était représenté par une simple liste de *mlvalue*, mais cela n'a pas fonctionné.

Un premier problème a été rencontré lors de l'implémentation de l'instruction APPLY, lorsque qu'un environnement vide devait être empilé dans la pile (*mlvalue list*). La concaténation d'une liste vide avec une autre liste faisait disparaître la liste vide ce qui faussait l'état de la pile. Pour garder cet environnement dans la pile, nous avons donc pensé à ajouter un constructeur constant None dans la définition du type mlvalue. Une liste avec None à l'intérieur représentait donc un environnement vide. Quelques premiers tests qui ne manipulaient pas l'environnement fonctionnaient avec cette version mais un deuxième problème s'est présenté assez rapidement. Nous pensions de l'environnement comme étant un tout, c'est-à-dire une valeur, et que lorsqu'on l'empilait dans la pile parmi d'autres éléments, cet environnement serait gardé comme un tout. Or, nous n'avions pas pris en compte le fait que la concaténation aplatissait les listes, la liste représentant l'environnement n'était plus une liste, mais juste une suite de mlvalue parmi d'autres dans la pile. Il fallait donc trouver un moyen d'envelopper l'environnement, pour permettre de la stocker dans la pile. Etant donné que la pile ne prend que des mlvalue, il a été évident d'ajouter dans le type un constructeur Env pour envelopper la liste de mlvalue de l'environnement.

Enfin le constructeur Bloc a été ajouté pour l'extension sur les blocs. il prend un *mlvalue array* en argument. Nous l'avions d'abord implémenté avec une liste, mais les instructions SETFIELD, SETVECTITEM et ASSIGN nous ont très vite arrêtés et nous a contraints à changer notre implémentation pour l'array compte tenu du caractère mutable des blocs et des effets de bord que cela implique.

### 3.2 Registres

La mini-ZAM contient au total 7 registres, ils sont définis comme variables globales dans *minizam.ml*.

En Ocaml, il n'y a pas de déclaration de variable, lorsqu'on introduit une variable, on doit lui affecter directement une valeur pour que le compilateur infère le type. Pour que la compilation se passe correctement, nous devons donc lui préciser le type des références. Pour le registre stack par exemple, on l'initialisait par une ref sur une liste vide, le compilateur inférait le type `'_weak1 list ref`, ce qui n'était pas bon. C'est pourquoi nous avons indiqué clairement le type des variables telles que prog, stack et env.

Le registre prog permet de stocker le programme à exécuter. Nous avons choisi de représenter chaque instruction par un enregistrement. Cela prend moins de place en mémoire par rapport à un n-uplet. De plus on peut manipuler les champs directement par leur nom, ce qui est rapide et plus facile. Nous avons donc déclaré le type de cet enregistrement dans le fichier *parser.ml* sous le nom de *triplet*. Prog est une liste de triplets. Un triplet est composé d'un champs *label*, *instr* et *args*. Le label est de type *string option* car il peut ne pas y avoir de label. L'instruction est de type *string*, juste pour contenir le nom de celui-ci. Et enfin, les arguments sont stockés dans une liste de string, une liste vide signifie qu'il n'y a pas d'argument dans l'instruction. Un programme est "parsé", puis stocké dans prog, on peut ensuite passer la passe dessus, qui a la possibilité de modifier des instructions, c'est pourquoi prog est une référence.

Le registre stack est une référence sur une liste de *mlvalue*, la pile doit être modifiable. Il existe un module Stack en Ocaml, mais nous avons préféré utiliser le module List. Les deux modules présentent chacun ses avantages et inconvénients, mais les fonctions de List répondaient mieux à nos besoins, et le seul inconvénient qu'il présentait était le manque de la fonction pop. Nous l'avons donc implémenté nous même dans avec la fonction *depile n* qui dépile directement de la stack n éléments, et retourne ces n *mlvalue* dans une liste.

Grâce à l'introduction du constructeur Env, le registre env est donc simplement une référence vers une *mlvalue*, ce qui nous facilite beaucoup de chose, mais cela induit une utilisation intense du motif de filtrage, pour récupérer la valeur d'un environnement.

Le registre accu est une référence sur une valeur de type *mlvalue*, il est initialisé au début à Entier(0). Enfin, les registres restant : pc, extra\_args, et trap\_sp sont de simples références sur un entier.

### 3.3 Instructions

Au niveau de l'implémentation des instructions, nous n'avons eu spécialement de soucis. Comme vous pourrez le constater dans le code source, nous nous sommes beaucoup appuyés sur le motif de filtrage étant donné les différents constructeurs pour le type *mlvalue*. L'implémentation des instructions nous a permis de détecter nos erreurs concernant les choix pour le type de *mlvalue* et des registres.

## 4 Optimisation APPTERM : comparaison de la pile

Pour les fichiers contenant la suite d'instructions APPLY n; RETURN m-n, nous avons comparé la taille maximum de la pile sans et avec la passe qui remplace cette suite par l'instruction APPTERM

n.m. On a notamment fait une version du test *fun\_appterm.txt* en remettant la suite APPLY et RETURN au lieu des APPTERM, le fichier se trouve dans le dossier jeu\_de\_test. Voici le tableau qui présente les différentes tailles maximales de la pile :

Fichier test	Taille de pile SANS passe	Taille de pile AVEC passe
appterm/facto_tailrec.txt	112	8
jeu_de_test/fun_appterm.txt	948	8
block_values/insertion_sort.txt	34	30
block_values/liste_iter	50	19
n-ary_funs/grab2.txt	12	8
n-ary_funs/grab4.txt	11	8
unary_funs/fun4-nooptim.txt	13	7

TABLE 1 – Taille maximale de la pile sans et avec la passe appterm selon différents tests.

On remarque effectivement une différence, particulièrement pour les tests dans appterm, cette différence est flagrante. En fait, plus il y a d'appels récursives avec APPLY RETURN, plus la taille de la pile augmente si on n'utilise pas l'instruction APPTERM à la place.

## 5 Nouveaux jeux d'essai

Dans le dossier jeu\_de\_test, nous avons écrit 3 nouveaux tests (sans compter le fun\_appterm.txt) pour la mini-ZAM. Vous y trouvez un fichier *.ml* ainsi qu'un fichier *.txt* pour chaque test.

Le test *op\_arithm* teste l'enchaînement des différentes opérations arithmétiques. *exception* teste une exception non levée. Et enfin le dernier, *map* teste les blocs avec de la récursivité.

## 6 Conclusion

L'implémentation de la mini-ZAM nous a permis de comprendre le fonctionnement d'un interprète bytecode fonctionnel, notamment au niveau du fonctionnement interne de la machine. Les sessions de débogage ont été enrichissantes, dans le sens où l'on devait simuler son évolution lorsque nous avions des erreurs. De plus, faire ce projet en Ocaml nous avons appris à mieux connaître ce langage, tout le côté fonctionnel typé static a été très convaincant au niveau de l'efficacité.