Chapitre 1

Conception

On a choisi OCaml pour les raisons expliquées dans la partie sur les contraintes.

Nous avons représenté l'IA de la fourmi comme un automate dont on peut voir les transitions comme une association de l'état courant de l'IA et de l'environnement de la fourmi qui produit la commande à exécuté au tour suivant.

Nous avons décidé de faire une IA propre à chaque fourmi, sans interaction entre les fourmis (cela reste possible mais avec une implémentation d'IA particulière).

En suivant cette logique, chaque fourmi à son environnement relatif (pas de notion d'environnement global).

Le déroulement du jeu du point de vue de notre programme consiste en une boucle qui actualise l'environnement de chaque fourmi (à partir du serveur), puis avec ces informations, demande à l'IA la commande à exécuter.

En ce qui concerne les zombies, nous avons simplement représenté les instructions pour pouvoir écrire du code assembleur un peu plus lisible.

Même si ce n'était pas une contrainte imposée, toute la partie IA est en fontionnel pur.

Chapitre 2

Contraintes

2.1 Contrainte 1 : Avoir un programme s'exécutant dans un navigateur web

2.1.1 Choix de la technologie

Nous avons décidé d'utiliser du Javascript afin d'implémenter cette première contrainte. Le Javascript a l'avantage d'apporter de la réactivité en terme d'User Expérience mais l'inconvénient que présente ce langage est l'absence de typage. C'est pourquoi un programme écrit en Javascript peut tout à fait lever une erreur lors de son utilisation, alors qu'un langage muni d'un typage fort aurait pu la détecter. De plus, tout notre projet a été implémenté en OCaml, il fallait donc utiliser une technologie qui nous permettrait d'utiliser ce langage pour cette première contrainte.

Nous avons donc décidé d'utiliser le compilateur de bytecode Ocaml vers du Javascript, *js_of_ocaml*. Cette technologie présente l'avantage de de contenir toutes les fonctionnalités qu'on trouve en Javascript accompagné du compilateur d'OCaml. En utilisant *js_of_ocaml*, on s'assure de garanties statique qui apporte de la persistance à notre application et plus particulièrement sur le Javascript généré. Le dernier point qu'on pourrait souligner est que son utilisation est tout à fait compatible avec OCaml et rend le déploiement de notre application possible.

2.1.2 Organisation

Nous avons découpé l'implémentation de cette première contrainte en plusieurs modules afin de leur attribuer un rôle bien précis. Chaque module

comporte une interface permettant à un client de comprendre son rôle et les fonctionnalités proposées par ce module. La liste suivante résume ce découpage :

1. Http : permet l'envoi de requêtes HTTP en utilisant les méthodes POST et GET.
2. IO : lecture du JSON et transformation vers des types OCaml nécessaire pour des parties de notre application.
3. Js_client : définition des comportements à adopter pour chaque élément graphique présents dans l'interface graphique (par exemple, que doit-il se passer si l'utilisateur clique sur le bouton « register »).
4. Js_client_ui : affichage de chaque élément sur l'interface graphique du client.
5. Main : Ajout de chaque bouton accompagné de leurs callback respectives. Ce module agrége toutes les fonctionnalités des modules précèdents pour permettre à l'utilisateur d'interagir avec l'application.

2.1.3 Problèmes rencontrés

Afin d'effectuer des requêtes HTTP, nous avons utilisé dans un premier temps *ocurl*, qui permet l'utilisation des primitives CURL en OCaml. Lorsque nous avons compilé l'ensemble de notre programme avec *js_ of_ ocaml*, ce dernier nous a indiqué l'absences de certaines primitives. En effet, *ocurl* utilise des librairies C qui ne peuvent être traduits en Javascript. L'alternative à ce problème fut d'utiliser le module XmlHttpRequest proposé par *js_ of_ ocaml*.

Le principal problème que nous avons rencontré est la récupération du cookie. Lorsque nous effectuons une authentification, le serveur doit placer le cookie dans l'entête de la réponse et il fut impossible de le récupérer. C'est assez problématique car l'API du projet impose qu'on soit connecté pour pouvoir l'utiliser. Un ticket a été ouverte sur *js_ of_ ocaml* afin de savoir si une solution existait à ce problème : nous avons été redirigé vers la documentation w3c de XmlHttpRequest, qui indique qu'il n'est pas possible de récupérer la valeur d'un cookie dans son entête. Même en désactivant quelques options de sécurité, il fut impossible de récuperer le cookie.

Dans les sources fournies, nous récupérons les cookies et les placons dans l'entête de chaque requête du client. Une solution pour s'assurer que la manière avec laquelle nous parsons le JSON est correcte fut d'utiliser l'exécutable fourni au début du projet (*antroit.sh* et *play-games.sh*). Nous récupérions le JSON pour le placer dans un fichier et de tester si nos fonctions

étaient le traitement spécifié correctement. C'est le seul moyen que nous avions de tester nos programmes.

2.2 Contrainte 2 : Rejouer une partie

Nous n'avons pas implémenté cette deuxième contrainte par manque de temps. Mais nous allons expliquer dans les grandes lignes comment nous aurions procédé.

Nous avons compris cette contrainte comme le fait de pouvoir exécuté exactement la même partie (offline) que jouée précédemment (online). Notre automate étant déterministe, pour une même entrée(état+ environnement) il y aura toujours la même sortie. Connaissant cette propriété, il suffit simplement de stocker à chaque tour le résultat de la commande *play* .Puis, quand on veux rejouer la partie, il suffit de relancer une partie en replaçant l'interface de communication par une simulation qui renvoi dans l'ordre les résultats stockés.