Machine Universelle et compilateur S-UM

Vincent Ha-Vinh
3305492

I. Description technique de la machine universelle

Choix de représentation des données :

- Un plateau de sable contenant 32 bits, on décide de le représenter par un unsigned int, qui est codé sur 4 octets (sur les architectures 32 bits).

- On représente un tableau de plateaux par :

```
typedef struct tableau_struct{
    uint taille_plateaux;
    uint* plateau;
}tableau;
```

  Car on a besoin ultérieurement de la taille d'un tableau déjà alloué pour pouvoir créer une copie de ce tableau (et donc allouer la même place mémoire à la copie), et pour recopier les instructions en itérant jusqu'au dernier élément.

  Une autre implémentation de type liste chaînée, en allouant les plateaux un par un lors d'une boucle itérant jusqu'à ce que le pointeur sur élément suivant soit NULL est aussi possible.
  Mais cela demande plus de maintenance pour l'allocation/désallocation et pour l'accès indexé à une case (opérateurs de modification/indice de tableau).

- A noter : un pointeur est codé sur 4 octets en C (architecture 32 bits).
  Les 8 registres à capacité d'un plateau (32bits) peuvent donc contenir aussi bien un plateau-instruction qu'un pointeur sur un tableau de plateaux.

Lecture des fichiers .um et chargement du programme .
Pour produire un tableau de plateaux contenant toutes les instructions d'un fichier .um, on procède de la sorte :

- On déclare un buffer de 4 int pour stocker les octets du plateau en cours de création sous la forme d'entiers.
  Chaque entier-octet aura donc 24 bits à 0 et 8 bits représentant l'octet lu à droite.
- On lit octet par octet le fichier .um, et on remplit le buffer.
- Une fois le buffer plein, on crée le plateau avec :

```
uint plateau = (bytes[3]) | (bytes[2]<<8) | (bytes[1]<<16) | (bytes[0]<<24);
```

  Les fichiers .um sont écrits en Big Endian, donc le 1er octet lu est celui de poids fort. Ces bits (8 bits) seront situés à gauche dans l'écriture binaire du plateau. On les décale donc de 24 bits vers la gauche. On fait de même pour le 2e et 3e octet, avec un décalage de 16 et 8 bits vers la gauche. Le 4e octet a déjà ses bits au bon endroit, à droite.
  On « fusionne » ensuite les entiers-octets en un seul entier ayant 4 x 8 bits représentant les 4 octets précédemment lus, ans le bon ordre (Big Endian).

Boucle de traitement du programme .
Chaque passe de cette boucle correspond à l'analyse d'un plateau (précédemment copié dans le tableau-programme parcouru) et à l'exécution de l'instruction encodée.

La phase d'analyse :

- On récupère le numéro de l'opérateur en extrayant les 4 bits de poids fort du plateau. Pour cela, on décale de 28 bits à droite les bits du plateau. Le nombre restant est alors de 28 bits à 0 et 4 bits représentant l'opérateur.

- Pour les plateaux codant une instruction avec un opérateur de 0 à 12 , on extrait les indexes des registres A B C à traiter :
  Le registre C est représenté par les 3 bits de poids faible du plateau.
  On applique donc un masque 000(29 fois)000111 aux bits du plateau, avec le ET logique, pour ne garder que la valeur des bits du plateaux qui correspondent à un bit 1 du masque appliqué .

```
id_A = (plat_act >> 6) & 7;
id_B = (plat_act >> 3) & 7;
id_C = (plat_act) & 7;
```

  7 = 000(29 fois)000111 en binaire

  On procède de même pour les registres B et A, mais en décalant auparavant de 3 et 6 bits à droite les bits du plateau, pour éliminer les bits de C avant d'appliquer le masque à B, et les bits de B et C avant d'appliquer le masque à A.

- Pour les plateaux codant une instruction d'opérateur d'affectation 13, on extrait l'index du registre A qui est représenté par les 3 bits de poids forts après ceux de l'opérateur,

```
id_A = (plat_act >> 25) & 7;
```

et la valeur à affecter qui est représentée par les 25 bits de poids faibles du plateau, en appliquant le masque 000000111(25 fois)111 aux bits du plateau.

```
value_A = (plat_act) & 33554431;
```

33554431 = 000000111(25 fois)111 en binaire.

La phase d'exécution de l'instruction :
(relevons les points intéressants du switch)

- L'opérateur 12, appelé Chargement du programme.
  Il charge le tableau pointé par la valeur du registre B à la place du programme principal, et place l'offset d'exécution au ième plateau du nouveau tableau-programme, où i est la valeur du registre C.

  Seulement, il s'avère après des tentatives de lancement de codex.umz et sandmark.umz (fichiers .um bien formés, données dans l'énoncé du concours), que la valeur du registre B peut être égale à 0. Or en C un pointeur qui stocke l'adresse 0 est en fait le pointeur sur NULL.
  Donc l'instruction demande de charger un tableau inexistant. Cela créera une erreur de l'affectation de l'accès à la taille du tableau inexistant.
  On suppose donc que si *registre[ic_C] = 0*, on n'effectuera pas de chargement de tableau.

  Après une nouvelle tentative de lancement de sandmark.umz, la UM indique « *LOADPROG off (low)* » avant de terminer.
  En fait, il s'avère que même si *registre[ic_C] = 0* et qu'on ne fait pas de chargement de tableau, on effectue quand même l'étape de déplacement de l'offset d'exécution au ième plateau. Seulement, ce sera le ième plateau du tableau prog d'origine, inchangé.
  Ce cas particulier correspond en fait à une instruction Jump !

- L'opérateur 10, Sortie.
  L'énoncé précise qu'il ne peut afficher que des valeurs comprises entre 0 et 255.
  Ces valeurs sont toutes codées sur un seul octet.
  On peut donc se servir de la fonction *fputc()*, qui permet d'écrire un character dans le buffer. Or un char est toujours codé sur un octet en C.
  Ainsi l'opérateur 10 permet d'afficher un char seulement.
  Dans un programme .um, l faudra boucler les plateaux d'opérateur 10 sur chaque char d'une chaine pour l'afficher.

Problème : Il est bien plus compliqué de créer des instruction dans un .um pour afficher un entier supérieur à 9 ! Puisqu'un entier est stocké sur 4 octets, mais que les octets composant cet entier ne correspondent absolument pas à un chiffre de l'entier chacun.

II. Compilateur SUM

La partie compilateur du projet est située dans le dossier sum/ .
Elle est implémentée en Ocaml et les outils ocamllex et ocamlyacc pour générer un lexer et parser à partir de la grammaire du langage SUM.

Les déclarations des différents nœuds de l'AST et leurs constructeurs sont dans le fichier **ast.ml**.

Le fichier **toSum.ml** est la phase de compilation où l'AST est parcouru et chaque instruction SUM d'un fichier *.sum* est transformée en des instructions UM encodées sur 32 bits et écrites dans un fichier *.um* de même nom.
On ouvre un flux d'entrée sur le fichier *test*.sum* passé en argument (ou *stdin* si pas d'argument) et un flux de sortie sur un fichier de nom *test*.um.*

Pour écrire une instruction UM, on utilise les fonctions *op13 opABC* :

- **op13 id_A valeur** : int→ int → unit
  Cette fonction prend 2 entiers en entrées, et écrit un plateau composé tel que :
  - les 4 bits de poids fort correspondent au numéro 0.
  - les 3 bits de poids fort suivants correspondent à l'entier *id_A*.
  - les 25 bits de poids faibles restants correspondent à l'entier *valeur.*

```
let plateau = ref 0 in
plateau := !plateau lor (13 lsl 28);
plateau := !plateau lor (id_A lsl 25);
plateau := !plateau lor (valeur land 33554431);
```

  Pour ce faire, on utilise un OU logique *lor* entre les bits du plateau (initialisés à 0) et :
  - les bits de *13* décalés à gauche *(lsl)* de 28 bits,
  - les bits de *id_A* décalés à gauche de 25 bits (en sachant que id_A < 14,pas de maques nécessaire pour supprimer les 4 bits de poids forts restants après décalage qui sont à 0)
  - les bits de *valeur* auquels on applique le masque 0000000111(25 fois)111

  Puis la fonction écrit sur le flux sortant le plateau créé en appelant *output_binary_int : output_channel → int → int* qui écrit l'entier en argument sur le flux de sortie au format binaire en Big Endian.

- **OpABC op id_A id_B id_C** : int → int → int → int

Ecrit tout plateau dont l'opérateur n'est pas spécial.
On raisonne de la même façon que précédemment, avec des décalages différents pour les entiers *id_A, id_B et id_C.*

```
let plateau = ref 0 in
plateau := !plateau lor (op lsl 28);
plateau := !plateau lor (id_A lsl 6);
plateau := !plateau lor (id_B lsl 3);
plateau := !plateau lor (id_C);
```

Ensuite, on définit les fonctions *sum_nomNoeudAst* qui contiennent différents cas d'exécution selon si leur argument est tel ou tel constructeur (les types nœud d'AST sont variants polymorphes !)
Ce sont ces fonctions qui contiennent les **schémas de compilation** de notre compilateur.

Voici les nœuds AST qui ont été implémentées dans le compilateur :

```
type expr =
    ASTnum of int

type stmt =
      ASTprintExpr of expr
    | ASTprintChaine of string

type prog =
    ASTprog of stmt
```

(Un AST commencera forcément par un nœud prog)

Par exemple, la fonction sum_prog est :

```
let rec sum_prog x =
    match x with
          ASTprog a ->
            (
                sum_stmt a;
                opABC 7 0 0 0
            )
```

Elle utilise une autre fonction sum_stmt, dont une partie du corps est :

```
| ASTprintChaine a ->
    (
        for  i = 1 to (String.length a)-2 do
            op13 7 (Char.code (String.get a i));
            opABC 10 0 0 7;
        done
    )
```

Ainsi on peut en déduire le schéma de compilation pour l'instruction SUM print chaine :

[**op13 7 ascii(chaine[i])**, */*instr UM d'affectation d'une valeur dans un registre*/*
opABC 10 0 0 7] */* instr UM d'affichage console de la aleur dans un registre*/*

*on répète ces 2 instr * le nombre de char dans la chaine à afficher*

On finira toujours le programme par l'instruction UM **opABC 7 0 0 0** qui est un return (implicite dans le langage SUM, explicite dans le langage UM)

Pour l'instant, seules les instructions SUM **print num** et **print chaine** ont été implémentées.

III. Architecture du Système de Tests :

Les différents tests sont dans le dossier tests/
Ils sont de la forme :

- test-instrTestée.sum :
  programme SUM contenant une instruction SUM à tester.

- test-instrTestée.output :
  Sortie attendue par la machine universelle.

- test-instrTestée.um :
  binaire UM produit par le compilateur toSum et lisible par la machine universelle.

- test-instrTestée.um.out :
  Sortie produite lors de l'interprétation du binaire .um par la machine universelle.

Un script automatisant les tests (*auto.sh*) été mis à la racine du projet .
Il s'occupe de :
- compiler les sources des dossiers um/ et sum/,
- déplacer les deux exécutables produits dans le dossier test/
- lancer le compilateur toSum sur tous les fichiers test*.sum
- lancer la UM um sur tous les fichiers test*.um produits précédemment.
- comparer les fichiers test*.um.out aux fichiers test*.output
- effacer les fichiers produits pendant cette automatisation de tache (optionnel)

Dans le dossier tests/ sont aussi disponibles codex.umz et sandmark.umz, fichiers binaires exécutables par toute machine universelle fonctionnelle.