Projet C++

Abstract VM

Koalab koala@epitech.eu

Abstract: Le but de ce projet est de creer une machine virtuelle simple capable d'interpreter des programmes ecrits dans un langage assembleur simplifie.

Table des matières

Chapitre I

Une machine

Qu'elle soit virtuelle ou pas, une machine, telle que celle qui vous sert a lire ce sujet, a une architecture particuliere. La seule veritable difference entre une machine dite "virtuelle" et une machine physique est que la machine physique utilise de veritables composants electroniques pour fonctionner alors que la machine virtuelle emule ces composants avec un programme.

Une machine virtuelle n'est ni plus ni moins qu'un programme simulant une machine physique ou une autre machine virtuelle. Toutefois, il est clair qu'une machine virtuelle emulant une machine physique telle qu'un "ordinateur" au sens usuel est un programme excessivement complique demandant une colossale experience en programmation et de profondes connaissances en architecture.

Pour ce projet, nous nous limiterons a une machine virtuelle extremement simple : Executer des programmes arithmetiques basiques ecrits dans un langage assembleur tout aussi basique. Si vous voulez une idee de ce dont votre programme sera capable une fois termine, tapez donc la commande `man dc` dans votre shell.

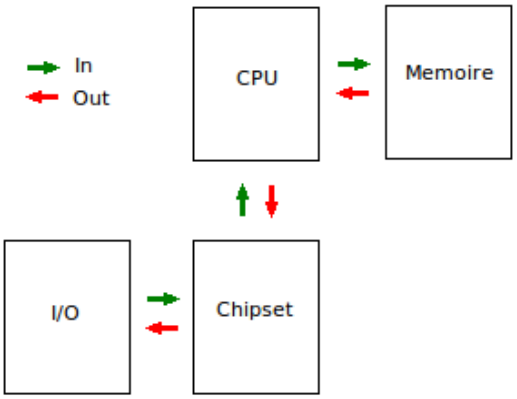
Chapitre II

Architecture

Notre machine aura donc une architecture classique. Toutefois, vous etes en droit de vous demander ce a quoi correspond une architecture classique.

Il n'y a pas de reponse simple a cette question. En effet, tout depend de la precision que vous voulez adopter et du type de probleme que vous voulez considerer. Chaque "organe" d'une machine peut-etre traduit en un programme plus ou moins complexe, et la complexite de ce programme est en fonction de ce que votre machine va servir a faire au final. Prenons par exemple la memoire. Nous sommes d'accord qu'entre une machine virtuelle capable de faire tourner un systeme d'exploitation tel que Linux ou Windows et une machine virtuelle de CoreWar : La complexite de l'emulation de la memoire de la machine sera completement differente !

Nous allons donc considerer l'architecture suivante :



Bien que cette architecture soit loin d'etre precise, elle est correcte et peut servir de base a l'architecture de la machine de ce projet.
Toutefois, posez-vous serieusement la question de savoir si elle est suffisante. Il est evident qu'il faut augmenter la precision, mais peut-etre que certains composants manquent. Ceci n'est pas une question fermee...
L'essentiel est que vous reflechissiez.

Quoi que vous decidiez, allez donc faire un tour a ces adresses :

1. http://fr.wikipedia.org/wiki/Processeur

2. http://fr.wikipedia.org/wiki/Chipset

3. http://fr.wikipedia.org/wiki/M%C3%A9moire_%28informatique%29

4. http://fr.wikipedia.org/wiki/Entr%C3%A9es-sorties

Chapitre III

Le projet

AbstractVM est une machine a pile capable de calculer des expressions arithmetiques simples. Les expressions arithmetiques sont donnees a la machine sous forme de programmes assembleur simples.

III.1 L'assembleur

III.1.1 Exemple

Un exemple valant mieux que toutes les explications du monde, voici un exemple de programme assembleur que votre machine doit pouvoir executer :

```
; -------------
; exemple.avm -
; -------------

push int32(42)
push int32(33)

add

push float(44.55)

mul

push double(42.42)
push int32(42)

dump

pop

assert double(42.42)

exit
```

III.1.2 Description

Comme tout langage assembleur, celui d'AbstractVM est compose d'une suite d'instructions avec une seule instruction par ligne. Difference notable avec d'autres langages assembleur, celui d'AbstractVM est type.

- Commentaires : Les commentaires debutent avec un ';' et se terminent a la fin d'une ligne. Un commentaire peut se trouver indifferemment en debut de ligne ou apres une instruction.
- **push *v*** : Empile la valeur v au sommet de la pile. La valeur v a forcement l'une des formes suivantes :
  - **int8(*n*)** : Cree un entier 8 bits ayant pour valeur n.
  - **int16(*n*)** : Cree un entier 16 bits ayant pour valeur n.
  - **int32(*n*)** : Cree un entier 32 bits ayant pour valeur n.
  - **float(*z*)** : Cree un flottant ayant pour valeur z.
  - **double(*z*)** : Cree un double ayant pour valeur z.
- **pop** : Depile la valeur au sommet de la pile. Si la pile est vide, l'execution du programme doit s'arreter en erreur.
- **dump** : Affiche chaque valeur sur la pile de la plus recente a la plus ancienne, SANS MODIFIER la pile. Chaque valeur est separee de la suivante par un retour a la ligne.
- **assert *v*** : Verifie que la valeur au sommet de la pile est bien egale a celle passee en parametre de cette instruction. Si ce n'est pas le cas, l'execution du programme doit s'arreter en erreur. La valeur v a bien sur la meme forme que celles passees en parametres a l'instruction **push**.
- **add** : Depile les deux premieres valeurs sur la pile, les additionne, puis empile le resultat. Si le nombre de valeurs sur la pile est strictement inferieur a 2, l'execution du programme doit s'arreter en erreur.
- **sub** : Depile les deux premieres valeurs sur la pile, les soustrait, puis empile le resultat. Si le nombre de valeurs sur la pile est strictement inferieur a 2, l'execution du programme doit s'arreter en erreur.
- **mul** : Depile les deux premieres valeurs sur la pile, les multiplie, puis empile le resultat. Si le nombre de valeurs sur la pile est strictement inferieur a 2, l'execution du programme doit s'arreter en erreur.
- **div** : Depile les deux premieres valeurs sur la pile, les divise, puis empile le resultat. Si le nombre de valeurs sur la pile est strictement inferieur a 2, l'execution du programme doit s'arreter en erreur. De plus, si le diviseur est egal a 0, l'execution du programme doit s'arreter en erreur egalement.
- **mod** : Depile les deux premieres valeurs sur la pile, calcule leur modulo, puis empile le resultat. Si le nombre de valeurs sur la pile est strictement inferieur a 2, l'execution du programme doit s'arreter en erreur. De plus, si le diviseur est egal a 0, l'execution du programme doit s'arreter en erreur egalement.

- **print** : S'assure que la valeur au sommet de la pile est un entier 8 bits (Dans le cas contraire, voir l'instruction `assert`), puis l'interprete comme une valeur ASCII et affiche le caractere correspondant sur la sortie standard.
- **exit** : Termine l'execution du programme en cours. Si cette instruction n'apparait pas alors que toutes les instructions ont ete executees, l'execution doit s'arreter en erreur.

`Dans le cas des operations non commutatives on considera pour la pile suivante : v1 sur v2 sur reste_pile, le calcul en notation infixe suivant : v2 op v1.`

Dans le cas ou un calcul implique deux operandes de types differents, la valeur de retour sera du type de l'operande la plus precise. Notez que dans un soucis d'extensibilite de votre machine, la question de la gestion de la precision peut ne pas etre triviale. Nous verrons cela en detail plus loin dans ce document.

III.1.3 Grammaire

Le langage assembleur de l'AbstractVM est genere a partir de la grammaire suivante (# correspond a la fin de l'entree, pas au charactere '#') :

```
1   S := [INSTR SEP]* #
2
3   INSTR :=
4       push VALEUR
5     | pop
6     | dump
7     | assert VALEUR
8     | add
9     | sub
10    | mul
11    | div
12    | mod
13    | print
14    | exit
15
16  VALEUR :=
17     int8(N)
18   | int16(N)
19   | int32(N)
20   | float(Z)
21   | double(Z)
22
23  N := [-]?[0..9]+
24
25  Z := [-]?[0..9]+.[0..9]+
26
27  SEP := '\n'
```

III.1.4 Erreurs

Dans le cas ou l'un des cas suivants se produit, AbstractVM doit lever une exception et arreter l'execution du programme proprement. Il est interdit de lever des exceptions

de type scalaire et vos classes d'exception devront heriter de `std::exception` de la `STL`.

- Le programme assembleur comporte une ou plusieurs fautes lexicales ou syntaxiques
- Une instruction est inconnue
- Overflow sur une valeur
- Underflow sur une valeur
- Instruction pop sur une pile vide
- Division/modulo par 0
- Le programme ne comporte pas d'instruction exit
- Une instruction assert n'est pas verifiee
- La pile comporte strictement moins de deux valeurs lors de l'execution d'un instruction arithmetique.

Peut-etre existe il d'autres erreurs, quoiqu'il en soit, votre machine ne doit jamais planter (segfault, bus error, boucle infinie, ...).

III.1.5 Execution

Votre machine doit pouvoir etre capable d'executer des programmes depuis des fichiers passes en parametres ou depuis l'entree standard. Dans le cas d'une lecture depuis l'entree standard, la fin du programme sera marquee par le symbole special ";;".

Attention a ne pas avoir de conflits lors des analyses lexicale ou syntaxique entre ";;" (fin d'un programme lu sur l'entree standard) et ";" (debut d'un commentaire).

Voyons ensemble quelques exemples d'executions :

```
>./avm
push int32(2)
push int32(3)
add
assert int32(5)
dump
exit
;;
5
>
```

```
>cat sample.avm
; -------------
```

```
; sample.avm -
; -------------

push int32(42)
push int32(33)
add
push float(44.55)
mul
push double(42.42)
push int32(42)
dump
pop
assert double(42.42)
exit
>./avm ./sample.avm
42
42.42
3341.25
>
```

```
>./avm
pop
;;
Line 1 : Error : Pop on empty stack
>
```

Le message d'erreur est donne a titre d'exemple, vous etes libres de mettre les votres a la place.

Chapitre IV

Considerations techniques

Afin de vous aider dans votre developpement, vous allez devoir respecter les consignes suivantes. Pour chaque consigne, reflechissez bien a pourquoi cette consigne vous a ete donnee. Aucune n'est la au hasard ou pour vous ennuyer.

IV.1 L'interface IOperand

Toutes vos classes d'operandes devront IMPERATIVEMENT implementer l'interface IOperand ci-desssous :

```
1  class IOperand
2  {
3  public:
4
5    virtual std::string const & toString() const = 0; // Renvoie une string reprensentant l'instance
6
7    virtual int getPrecision() const = 0; // Renvoie la precision du type de l'instance
8    virtual eOperandType getType() const = 0; // Renvoie le type de l'instance. Voir plus bas
9
10   virtual IOperand * operator+(const IOperand &rhs) const = 0; // Somme
11   virtual IOperand * operator-(const IOperand &rhs) const = 0; // Difference
12   virtual IOperand * operator*(const IOperand &rhs) const = 0; // Produit
13   virtual IOperand * operator/(const IOperand &rhs) const = 0; // Quotient
14   virtual IOperand * operator%(const IOperand &rhs) const = 0; // Modulo
15
16   virtual ~IOperand() {}
17 };
```

Les classes d'operandes implementant l'interface IOperand doivent etre les suivantes :

- **Int8** : Representation d'un entier signe code sur 8bits.
- **Int16** : Representation d'un entier signe code sur 16bits.
- **Int32** : Representation d'un entier signe code sur 32bits.
- **Float** : Representation d'un flottant.
- **Double** : Representation d'un double.

Important : Il est INTERDIT de manipuler des pointeurs ou des references sur chacune de ces 5 classes. Vous n'avez le droit de manipuler QUE des pointeurs sur des IOperand.

Vu les similarites entre les classes d'operandes, il pourrait etre pertinent d'utiliser des classes templates. Toutefois, ce n'est pas obligatoire.

IV.2 Creation d'un nouvel IOperand

Vous devez ecrire une fonction membre d'une classe **pertinente** de votre machine qui vous permettra de creer de nouveaux operandes de maniere generique. Cette fonction membre devra avoir le prototype suivant :

```
1    IOperand * createOperand(eOperandType type, const std::string & value);
```

Le type `eOperandType` est un enum pouvant prendre les valeurs suivantes :

- Int8
- Int16
- Int32
- Float
- Double

En fonction de la valeur de l'enum passe en parametre, la fonction membre `createOperand` creera un nouveau `IOperand` en faisant appel a l'une des fonctions membres `privees` suivantes :

```
1    IOperand * createInt8(const std::string & value);
2    IOperand * createInt16(const std::string & value);
3    IOperand * createInt32(const std::string & value);
4    IOperand * createFloat(const std::string & value);
5    IOperand * createDouble(const std::string & value);
```

Pour selectionner la bonne fonction membre utilisee pour la creation du nouvel `IOperand`, vous DEVEZ imperativement creer et utiliser un tableau (ici le `vector` n'a que peu d'interet) de pointeurs sur fonctions membres indexe par les valeurs de l'enum.

IV.3 La precision

Lorsqu'une operation a lieu entre deux operandes du meme type reel, pas de probleme, mais qu'en est il lorsque les types reels sont differents ?

La methode usuelle est d'ordonner les types entre eux par ordre de precision. Dans le cas de notre machine, cela donne l'ordre suivant :

```
1    Int8 < Int16 < Int32 < Float < Double
```

Pour utiliser cet ordre dans notre machine on peut associer un entier a chaque type en conservant l'ordre, via un enum par exemple.

La methode pure `getPrecision` de l'interface `IOperand` permet de connaitre la precision de l'operande consideree. Lors d'une operation mettant en cause deux operandes de types differents, en comparant leurs precisions, on peut determiner quel sera le type du resultat de l'operation.
Dans ce projet, on considerera que le type renvoye est toujours le plus precis des deux.

IV.4 La pile

AbstractVM etant une machine a pile, elle possedera donc un conteneur se comportant comme une pile, voire directement une pile. Le choix vous parait evident maintenant mais peut-etre decouvrirez-vous une subtilite qui vous fera vous reposer la question plus tard.

Votre conteneur ne doit contenir **QUE** des pointeurs sur le type abstrait IOperand ! Il est **INTERDIT** de stocker des types reels d'operandes dans votre conteneur.

Chapitre V

Consignes

Vous etes globalement libres de faire l'implementation que vous voulez. Cependant, il y'a quelques restrictions :

- Toute bibliotheque a part la `STL` est explicitement interdite.
- Vous devez utiliser la `STL` le plus possible, on doit donc trouver dans votre projet au moins un conteneur et une classe d'exception issues de la `STL`. L'utilisation d'au moins un algorithme issu de la `STL` sera apprecie. Faites bien attention.
- Les seules et uniques fonctions de la `libc` autorisees sont celle qui encapsulent les appels systemes, et qui n'ont pas d'equivalent C++.
- "split", "strtowordtab", et compagnie ne doivent **pas** etre utilises pour parser du code (meme de l'assembleur !). Nous vous fournissons une grammaire, utilisez la !
- Toute valeur passee par copie plutot que par reference ou par pointeur doit etre justifiee, sinon vous perdrez des points.
- Toute valeur non `const` passee en parametre doit etre justifiee, sinon vous perdrez des points.
- Toute fonction membre ou methode ne modifiant pas l'instance courante n'etant pas `const` doit etre justifiee, sinon vous perdrez des points.
- Il n'existe pas de norme en `C++`. Cependant, tout code que nous jugerons illisible ou trop sale pourra etre sanctionne. Soyez serieux !
- Gardez un oeil sur ce sujet regulierement car il est succeptible d'etre modifie.

Chapitre VI

Consignes de rendu

Vous devez rendre votre projet sur le depot mis a votre disposition par Epitech. Le nom du depot a employer est cpp_abstractvm.

Vos depots seront clones a l'heure exacte de la fin du projet, intranet `Epitech` faisant foi.

Seul le code present sur votre depot sera evalue lors de la soutenance.

Bon courage !