Mécanisme d'Émulation de Mémoire

3h (avec documents)

Année 2021-2022

Préambule

- Lorsqu'ils ne sont pas déjà fournis, les **contrats et les tests unitaires doivent être donnés**. Il vous est toujours possible d'ajouter des tests unitaires à des tests déjà existants si vous en ressentez le besoin. Les tests unitaires additionnels **pertinents** seront valorisés.
- Pensez à tester régulièrement ce que vous écrivez. Vous pouvez commenter des morceaux de code qui génèrent des erreurs.
- Interdiction formelle de modifier les signatures (nom + type) fournies. Les vérifications seront automatisées !
- Le code rendu doit **impérativement compiler** (si une partie ne compile pas, mettez-la en commentaires).
- Les parties 2 et 3 sont indépendantes ; lisez bien le sujet !

1 Introduction

L'**émulation** consiste à réaliser des fonctionnalités matérielles à l'aide d'un logiciel. Un processeur est alors remplacé (« émulé ») par un bout de programme qui se charge d'interpréter des instructions, et la mémoire de la machine est simulée par des structures de données, accompagnées de diverses fonctionnalités qui permettent au programme-processeur d'interagir avec (API).
Il existe un grand nombre de structures de données pour représenter la mémoire et son fonctionnement. Dans le cadre de ce sujet, notre but sera d'étudier leurs similarités et leurs différences, et notamment leurs efficacités en terme de mémoire physique occupée et de temps d'accès écriture/lecture.

1.1 Spécification

Les mémoires que nous manipuleront sont spécifiées dans l'interface `Memory`, définie dans le fichier `mem.ml`. Formellement, une mémoire est définie comme un objet qui associe des *adresses* (sous forme d'entiers) à des *valeurs* (sous forme d'octets, donc ici de `char`). Il est possible de lire la valeur située à une adresse donnée (`read`), d'écrire une valeur à une adresse donnée (`write`), et de remettre à zéro/créer une nouvelle mémoire (`clear`).
Une mémoire est caractérisée avant tout par la *taille de son bus d'adressage* (`bussize`), autrement dit le nombre de bits des adresses auxquelles on peut accéder. Dans une optique de profilage, il est aussi intéressant de pouvoir calculer le nombre maximal de valeurs que la mémoire peut stocker (`size`), ainsi que son « occupation », c'est-à-dire le nombre de valeurs effectivement stockées (`busyness`).
Par exemple la figure 1 représente une mémoire avec un bus de taille 6. Les adresses seront donc comprissent entre 000000 et 111111 en base 2 et donc entre 0 et 63 en base 10.
Enfin, et dans un but d'évaluation de l'efficacité de la mémoire, on veut pouvoir calculer la taille que la mémoire émulée prend en mémoire physique, autrement dit la taille de la structure de donnée dans la RAM (`allocsize`).

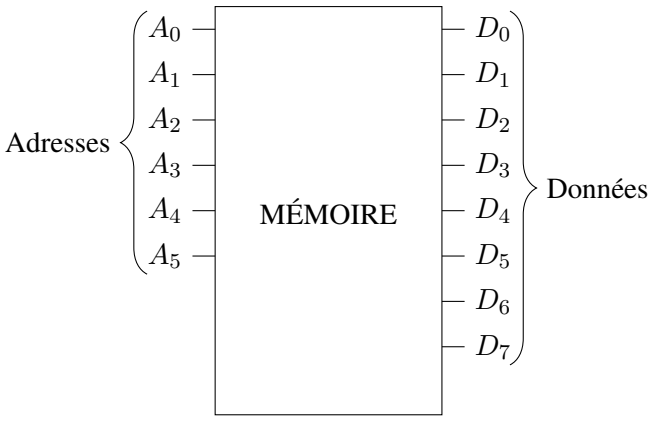

FIGURE 1 – Représentation schématique d'un module mémoire (bus d'adressage : 6 bits)

1.2 Tests et profilage

Nous mettons à disposition un foncteur de test (`MemoryTest` dans le fichier `test.ml`) qui permet de tester les modules de mémoire proposés.
Lorsque vous aurez complété une implémentation de `Memory`, décommentez la ligne correspondante dans `test.ml` afin de pouvoir lancer les tests associés avec `dune runtest`.

En plus du test des fonctionnalités, le but du BE est avant tout de comparer l'efficacité de différentes implémentations de mémoire. Dans ce but, nous fournissons un outil de profilage ou *benchmark* (foncteur `MemoryBench` dans le fichier `bench.ml`) qui effectue un certain nombre de mesure de temps et d'occupation mémoire pour chaque module écrit.
Une fois une implémentation complétée, vous pouvez l'ajouter au benchmark en décommentant les lignes correspondantes de `bench.ml` (instantiation du module, vers la ligne 117, et appelle à `dobench` dans la fonction `main`, vers la ligne 124).
La commande `dune build` permet de construire l'exécutable `bench.exe`. Il suffit alors d'utiliser la commande `dune exec ./bench.exe` pour lancer le benchmark. L'exécution peut prendre quelques dizaines de secondes.

1.3 Autres fichiers et architecture du projet

Le fichier `util.ml` contient deux petits utilitaires pratiques pour l'écriture de vos programmes : la fonction `pow2` qui permet de calculer rapidement 2^n, et la constante `_0` qui représente la valeur 0 dans la mémoire [1].
Pour rappel, les mémoires que nous manipuleront sont spécifiées dans l'interface `Memory`, définie dans le fichier `mem.ml`. Une implantation simpliste et peu efficace d'une mémoire à base de listes vous est fournie en exemple dans le fichier `listmem.ml`. Deux implantations vous serons demandées :

- à base de liste associative (adresse, valeur en mémoire) - fichier `assocmem.ml`
- à base de *bit tree* (structure arborescente définie dans le fichier `btree.ml`) - fichier `treemem.ml`

2 Une implémentation simpliste : des listes associatives

Puisque le but de notre mémoire est d'associer des valeurs à des adresses, nous nous proposons d'utiliser une *liste associative*, autrement dit une liste de couples clef-valeur, où la clef sera en fait l'adresse, et la valeur la chose stockée dans la mémoire à cette adresse.
Si une clef n'est pas dans la liste, on considérera que la valeur associée est la valeur par défaut, donc `_0`.

1. Nous utilisons le type `char` pour représenter les valeurs/octets en mémoire. Contrairement à certains autres langages (C notamment), le type `char` n'est pas compatible avec le type des entiers en OCaml, ce qui signifie que l'on ne peut pas écrire `0` pour représenter la valeur 0 de type caractère, d'où cette constante.

2.1 Manipulation sur les listes associatives

Avant de nous pencher sur l'implémentation de `Memory`, écrivons d'abord quelques fonctions générales sur les listes associatives qui nous faciliteront la vie.

▷ **Exercice 1** *Dans le préambule du fichier* `assocmem.ml` *:*

1. *Écrire le contrat, les tests unitaires et le corps de la fonction* `get_assoc`*, qui retourne la valeur associée à la clef* e *dans la liste* l*, ou la valeur fournie* def *si la clef n'existe pas.*
2. *Écrire le contrat, les tests unitaires et le corps de la fonction* `set_assoc`*, qui remplace la valeur associée à la clef* e *dans la liste* l *par* x*, ou ajoute le couple* (e, x) *si la clef n'existe pas déjà.*

2.2 Implémentation de `Memory`

Nous nous intéressons maintenant à l'implémentation `AssocMemory` de `Memory` avec les listes associatives (deuxième moitié du fichier `assocmem.ml`).

▷ **Exercice 2** *Sachant que l'on veut une liste qui associe des adresses (entiers) à des valeurs (caractères), quel type allons-nous utiliser ? Renseigner le type* `mem_type` *en conséquence.*

▷ **Exercice 3** *Notre mémoire est essentiellement une liste de couples ; sachant que chaque couple occupe 2 unités de mémoire physique (une pour l'adresse et une pour la valeur), quelle taille prend la liste en mémoire ? En déduire l'implémentation de* `allocsize`*. (L'utilisation de fonctions du module* `List` *d'OCaml est fortement conseillée)*

▷ **Exercice 4** *Écrire l'implémentation de* `busyness`*, qui compte les valeurs présentes dans la mémoire* ***qui ne sont pas nulles****. (L'utilisation de fonctions du module* `List` *d'OCaml est fortement conseillée)*

▷ **Exercice 5** *Comment représenter une mémoire vide à l'aide d'une liste ? En déduire l'implémentation de* `clear`*.*

▷ **Exercice 6** *Compléter enfin les implémentations de* `read` *et* `write` *à l'aide des fonctions écrites ci-avant. (On prendra soin de bien respecter les contrats proposés dans* `Memory`*, notamment en ce qui concerne la gestion des erreurs !)*

Remarque : Pensez à tester votre implantation à l'aide des informations indiquées dans la section 1.2.

3 Utilisation de *bit trees*

L'utilisation de listes souffre de quelques problèmes d'efficacité, en particulier lorsque la mémoire est très dispersée (on dit aussi *fragmentée*). Pour pallier cela, nous proposons d'implémenter la mémoire sous la forme d'un *bit tree*, ce qui devrait grandement améliorer les temps d'accès, sans trop pénaliser le poids en mémoire physique.

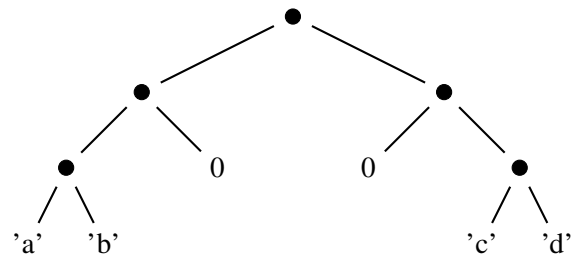

FIGURE 2 – Exemple de *bit tree* (correspondant à `bt1`)

Un *bit tree* est un arbre binaire dont les nœuds ne contiennent rien, mais dont les feuilles contiennent une valeur stockée dans l'arbre. Le chemin de la racine à la feuille donne son adresse bit à bit : un 0 pour chaque branche gauche, et un 1 pour chaque branche droite.
Lorsqu'un chemin n'existe pas complètement dans l'arbre, on se contente de retourner la valeur où il s'arrête.
Par exemple, dans l'arbre donnée à la figure 2 :

- La valeur 'a' est stockée à l'adresse `000` (gauche-gauche-gauche) ;
- La valeur 'c' est stockée à l'adresse `110` (droite-droite-gauche) ;
- Les adresses `010` et `011` contiennent la valeur 0 ;

On remarque que la *hauteur* de l'arbre (distance maximale entre la racine et une feuille) est au maximum égale au nombre de bits sur lesquels sont codés les adresses.

3.1 Type et opérations de base

On se place tout d'abord dans le fichier `btree.ml`, qui contient le type et les opérations sur les *bit tree*. Pensez à dé-commenter les sources (début du commentaire à la ligne 24).
Un bit tree est un arbre binaire dont les nœuds ne contiennent rien et les feuilles contiennent un caractère. Le type `btree` vous est donné.

▷ **Exercice 7** *Un bit tree est vide lorsqu'il ne contient que des 0. En déduire une expression de `empty_tree`.*

▷ **Exercice 8** *Écrire les fonctions de base sur les bit tree :*

1. *La fonction `height` qui calcule la hauteur d'un bit tree ;*
2. *La fonction `num_nodes` qui calcule le nombre de nœuds + feuilles d'un bit tree ;*
3. *La fonction `num_values` qui calcule le nombre de valeurs stockées dans l'arbre, i.e le nombre de feuilles de l'arbe qui ne sont pas égales à 0.*

3.2 Recherche dans l'arbre

Un *bit tree* permet d'accéder à des éléments à l'aide d'un « chemin », donné sous la forme d'un mot binaire. Dans ce mot, chaque 0 correspond à descendre dans le sous-arbre à gauche, et chaque 1 correspond à descendre dans le sous-arbre à droite.
Pour nous simplifier la vie, les fonctions d'accès à l'arbre prendront en paramètre des mots binaires, sous la forme de liste de 0 et de 1 ; cependant, comme l'interface `Memory` utilise des entiers pour les adresses, nous avons d'abord besoin d'une fonction qui transforme un entier en mot binaire (de taille fixe).

▷ **Exercice 9** *Comment obtenir le bit de poids faible (ou bit de parité) d'un nombre entier ? Comment obtenir les bits de poids fort, c'est à dire tous les bits excepté le bit de parité ?*
Écrire la fonction `bits` qui transforme un entier en liste de 0 et de 1 avec la taille désirée (si le mot ne peut pas être codé sur la taille donnée, on se contente de retourner les bits de poids faible). La liste ira du bit le plus faible (la puissance de deux la plus basse) au bit le plus fort.

▷ **Exercice 10** *Écrire la fonction `search`, qui parcourt un bit tree en suivant le chemin donné en paramètre, sous forme d'une liste de 0 et de 1.*

3.3 Écriture dans l'arbre

La difficulté d'écrire dans un *bit tree* vient du fait que certaines branches mènent à des « impasses ». Pour y ajouter une adresse, il faut donc créer un sous arbre entier, qui contient l'adresse, et des 0 partout ailleurs (exemple sur la figure 3).

▷ **Exercice 11** *Écrire la fonction `sprout`, qui construit le (sous-)bit tree qui contient l'élément donné à l'adresse donnée, et des 0 partout ailleurs.*

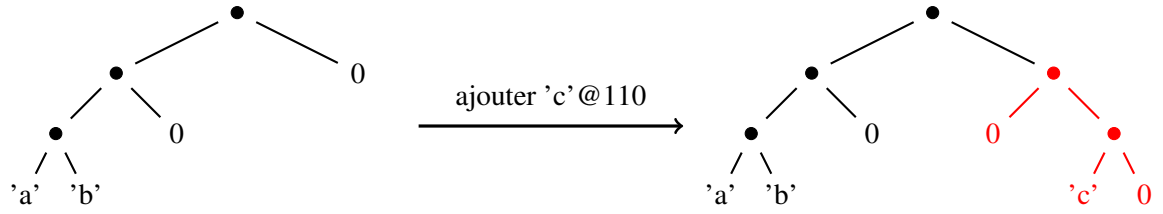

FIGURE 3 – Procédure d'ajout dans un *bit tree*

▷ **Exercice 12** *Écrire la fonction* `update`*, qui modifie la valeur associée à une adresse dans un bit tree. Si le chemin vers la valeur existe déjà, cette dernière est simplement mise à jour. Si le chemin n'existe pas, il faut construire le bon (sous-)bit tree.*

3.4 Implémentation de `Memory`

On s'intéresse maintenant à l'implémentation `TreeMemory` de `Memory` (dans le fichier `treemem.ml`).

▷ **Exercice 13** *Renseigner l'implémentation de* `Memory` *avec des bit tree, à l'aide des fonctions écrites dans* `btree.ml`*. On prendra soin de bien respecter les contrats donnés dans* `Memory`*, notamment pour les fonctions* `read` *et* `write`*!*
***Remarque** : Pensez à tester votre implantation à l'aide des informations indiquées dans la section 1.2.*