TP n° 5

Paradigmes et interprétation
Licence Informatique
Université Côte d'Azur

Le but de ce TP est d'introduire les pointeurs dans l'interpréteur `variable.rkt`. En effet, cet interpréteur manipule les adresses mémoire mais n'en donne pas l'accès à l'utilisateur. Les pointeurs sont un moyen d'accéder directement à la mémoire et de la modifier.

Les pointeurs

Un pointeur est la valeur d'une adresse mémoire. Les adresses mémoire sont représentées par le type `Location` qui est un alias pour le type `Number`. Il n'est donc pas nécessaire d'ajouter une nouvelle valeur à l'interpréteur : un pointeur sera simplement un `numV`. Une adresse mémoire est valide si c'est un entier strictement positif. On pourra utiliser la fonction suivante pour le TP.

```
(define (integer? n) (= n (floor n)))
```

On se donne trois nouvelles expressions pour manipuler les adresses mémoire.

```
<Exp> ::= ...
        | {address <Symbol>}
        | {content <Exp>}
        | {set-content! <Exp> <Exp>}
```

- L'expression `{address var}` s'évalue en l'adresse mémoire de la variable `var`.
- L'expression `{content loc}` renvoie le contenu de l'adresse mémoire `loc` ou une erreur `"segmentation fault"` si `loc` n'est pas une adresse valide.
- L'expression `{set-content! loc expr}` modifie le contenu de l'adresse mémoire `loc` en la valeur `val` de `expr`. L'évaluation de `{set-content! loc expr}` donne aussi la valeur `val`. Une erreur `"segmentation fault"` est renvoyée si `loc` n'est pas une adresse valide.

Complétez l'interpréteur pour qu'il prenne en charge ces nouvelles expressions.

Malloc

Ajoutez une expression `malloc` au langage qui crée dynamiquement un certain nombre de cellules. Lors de l'interprétation de l'expression `{malloc expr}`, si `expr` s'évalue à `(numV n)`, on crée `n` nouvelles cellules contiguës en mémoire initialisées à la valeur `(numV 0)`. L'expression s'évalue en l'adresse de la première d'entre elles. Si `expr` ne s'évalue pas en un entier strictement positif, on renvoie une erreur `"not a size"`.

```
(test (interp (parse `{let {[p {malloc 3}]} p}) mt-env mt-store)
      (v*s (numV 1) (list (cell 4 (numV 1)) ; addresse de p
                          (cell 3 (numV 0))
                          (cell 2 (numV 0))
                          (cell 1 (numV 0)))))
```

Free

On souhaite maintenant ajouter une instruction `free` au langage tel que `{free expr}` libère la mémoire à l'adresse `expr`. Une erreur `"not an allocated pointer"` est renvoyée si `expr` ne s'évalue pas en une adresse qui a été obtenue via un appel à `malloc`.

Il existe plusieurs méthodes pour vérifier qu'une adresse mémoire a bien été allouée par `malloc`. On se propose ici de gérer une liste de tous les pointeurs alloués. À tout moment, l'interpréteur devra avoir connaissance de cette liste. Pour ce faire, il faut modifier la représentation de la mémoire. Celle-ci contiendra comme précédemment la liste des cellules en mémoire mais aussi la liste des pointeurs créés par des instructions `malloc`. Un pointeur est représenté par l'adresse mémoire qui a été renvoyé par `malloc` lors de l'allocation ainsi que la taille de la mémoire allouée.

```
(define-type Store
  [store (storages : (Listof Storage))
         (pointers : (Listof Pointer))])

(define-type Pointer
  [pointer (loc : Location) (size : Number)])
```

Adaptez la constante `mt-store` ainsi que les fonctions `override-store`, `new-loc`, `max-address` et `fetch` en conséquence. Modifiez de même l'évaluation de l'expression `malloc` pour sauvegarder le pointeur produit.

Introduisez l'expression `free` dans le langage. Il faut retirer de la mémoire toutes les adresses libérées. La valeur retournée par l'expression est toujours `(numV 0)`. Prenez garde au fait que libérer deux fois le même pointeur doit produire une erreur `"not an allocated pointer"`.

```
(test (interp (parse `{let {[p {malloc 3}]} p}) mt-env mt-store)
      (v*s (numV 1) (store (list (cell 4 (numV 1))
                                 (cell 3 (numV 0))
                                 (cell 2 (numV 0))
                                 (cell 1 (numV 0)))
                           (list (pointer 1 3)))))

(test (interp (parse `{let {[p {malloc 3}]} {free p}})
              mt-env
              mt-store)
      (v*s (numV 0) (store (list (cell 4 (numV 1)))
                           empty)))
```