Rapport *Interim OS*

Marc Ducret Florentin Guth

13 décembre 2016

Table des matières

1 Structure de l'*OS* **2**
1.1 Architecture générale . . . 2
1.2 Liste des fichiers *C* . . . 2
1.3 Liste des fichiers *Minilisp* . . . 3
1.4 Liste des *builtins Minilisp* . . . 3

2 Allocation de la mémoire **5**
2.1 Représentation des valeurs *Minilisp* . . . 5
2.2 Gestion de la mémoire . . . 5
2.3 *Garbage collection* . . . 6

3 Compilation à la volée **6**
3.1 Fonctions auxiliaires . . . 6
3.2 La fonction `compile_expr` . . . 6

4 Améliorations apportées **7**
4.1 Optimisation du *garbage collecting* . . . 7
4.2 Système de tâches . . . 8
4.3 Analyse de la lenteur du rendu . . . 8

1 Structure de l'*OS*

1.1 Architecture générale

Interim OS est scindé en deux parties : un partie permettant de gérer précisément la mémoire, écrite en *C*, et une partie contenant la gestion des différents programmes (comme un éditeur, une console, ...).

La partie gérant la mémoire fonctionne de la manière suivante :

— On lit l'expression *Minilisp* donnée.

— On formate cette expression en une représentation plus structurée pour faciliter la compilation.

— On produit du code assembleur correspondant à l'exécution de l'expression donnée, que l'on stocke dans un fichier temporaire.

— Le code produit utilise des fonctions spéciales d'allocations qui permettent de faire fonctionner le ramasse-miettes pour libérer la mémoire lorsque c'est nécessaire.

— On exécute ce code assembleur.

— On affiche le résultat (qui est une valeur *Minilisp*).

1.2 Liste des fichiers *C*

Nom du fichier	Contenu
`strmap.[h|c]`	Opérations sur une table de hachage dont les clés sont des chaînes de caractères
`minilisp.h`	Contient la représentation mémoire des valeurs du *Minilisp*
`alloc.[h|c]`	Définition de l'environnement, des différents tas, du *garbage collector* et des fonctions d'allocations des différents types de cellules
`utf8.[h|c]`	Conversion entre chaînes de caractères standard et l'encodage *UTF-8*
`reader.[h|c]`	*Parser* de *minilisp*
`writer.[h|c]`	Fonctions pour écrire une valeur *Minilisp* dans un *buffer*
`stream.[h|c]`	Représentations des systèmes de fichiers, et fonctions pour les ouvrir, fermer, écrire...
`jit_x64.c`	Fonctions pour écrire de l'assembleur *x86-64*
`compiler_new.[h|c]`	Compile une expression *Minilisp* en assembleur, et initialise l'environnement avec les primitives *Minilisp*
`compiler_x64_hosted.c`	Compile l'expression en assembleur *x86-64*, l'exécute et renvoie le résultat de son exécution
`sledge.c`	Contient la fonction principale, qui ouvre un *channel* passé en argument, y lit une expression qu'il *parse* et exécute avant d'afficher le résultat

TABLE 1 – Liste des fichiers *C*

1.3 Liste des fichiers *Minilisp*

Nom du fichier	Contenu
`lib.l`	Fonctions de base sur les listes et les chaînes de caractères
`gfx.l`	Fonctions de base d'affichage de figures géométriques
`mouse.l`	Gestion de la souris comme système de fichiers
`net.l`	Communication sur internet (notamment par *IRC*) par un système de fichiers
`editor.l`	Fonctionnement de l'éditeur : affichage, gestion des touches pressées, ...
`repl.l`	Fonctionnement du *REPL (read-eval-print-loop)* : affichage, gestion de l'historique des commandes, ...
`paint.l`	Application de dessin ?
`shell.l`	Gestion des différentes tâches, ajout du logo, d'un éditeur et d'un *REPL*

TABLE 2 – Liste des fichiers *Minilisp*

1.4 Liste des *builtins Minilisp*

Signature	Effet
`(bitand a b)`	Et bit-à-bit
`(bitnot a b)`	Non bit-à-bit
`(bitor a b)`	Ou inclusif bit-à-bit
`(bitxor a b)`	Ou exclusif bit-à-bit
`(shl a b)`	Décalage logique vers la gauche
`(shr a b)`	Décalage logique vers la droite
`(+ a b)`	Addition
`(- a b)`	Soustraction
`(* a b)`	Multiplication
`(/ a b)`	Quotient de la division
`(mod a b)`	Reste de la division
`(gt a b)`	Test de supériorité
`(lt a b)`	Test d'infériorité
`(= a b)`	Test d'égalité

TABLE 3 – Liste des *builtins Minilisp* arithmético-logiques

Signature	Effet
`(def x v)`	Définit globalement `x` comme valant `v`
`(let x v)`	Définit `x` comme valant `v` (allocation locale sur la pile, qui pourra donc être *garbage-collectée*)
`(fn x1 .. xn r)`	Renvoie une fonction à `n` arguments qui renvoie `r`, un argument est soit un symbole soit de la forme `(symb struct_def)`, et `symb` sera alors de type `struct_def`
`(if b x y)`	Si `b` évalue à vrai, renvoie `x`, sinon `y` (qui doivent avoir le même type)
`(while b e)`	Exécute `e` tant que `b` est vrai
`(do x1 .. xn)`	Exécute `x1`, .., `xn` et renvoie `xn`
`(car l)`	Renvoie la tête de la liste `l`
`(cdr l)`	Renvoie la queue de la liste `l`
`(cons x l)`	Renvoie la liste `(x l)`
`(list x1 .. xn)`	Renvoie `(cons x1 (.. (cons xn-1 xn)..))`
`(struct s c1 x1 .. cn xn)`	Définit `s` comme une structure contenant `n` champs dont les noms sont `fi` et les valeurs par défaut `xi`
`(new s)`	Alloue et renvoie une structure de type `s`
`(sget s c)`	Renvoie la valeur du champ `c` de la structure `s`
`(sput s c v)`	Affecte la valeur `v` au champ `c` de la structure `s`

TABLE 4 – Liste des *builtins Minilisp* de contrôle

Signature	Effet
`(quote x)`	Renvoie l'adresse du symbole `x`
`(concat s t)`	Renvoie la concaténation des chaînes `s` et `t`
`(substr s a b)`	Renvoie une copie de la chaîne `s` entre `a` et `b`
`(get8 s i)`	Renvoie l'octet de la chaîne `s` situé en position `i`
`(get16 s i)`	Renvoie deux octets de la chaîne `s` à partir de `i`
`(get32 s i)`	Renvoie quatre octets de la chaîne `s` à partir `i`
`(put8 s i v)`	Modifie l'octet de la chaîne `s` situé en position `i`
`(put16 s i v)`	Modifie deux octets de la chaîne `s` à partir de `i`
`(put32 s i v)`	Modifie quatre octets de la chaîne `s` à partir `i`
`(alloc n)`	Alloue et renvoie `n` octets
`(alloc_str n)`	Alloue et renvoie une chaîne de `n` caractères
`(bytes_to_str b n)`	Alloue et renvoie une chaîne de `n` caractères obtenues depuis `b`
`|=|=| (size x)`	Renvoie la taille de `x` en mémoire
`(gc)`	Appelle la fonction `collect_garbage`
`(symbols)`	Renvoie la liste des symboles connus
`(debug)`	Devrait appeler `platform_debug` mais a été commenté

TABLE 5 – Liste des *builtins Minilisp* de gestion de la mémoire

Signature	Effet
`(write x b)`	Écrit la représentation de `x` dans le *buffer* `b`
`(read b)`	Lit le code *Minilisp* situé dans le *buffer* `b` et renvoie une valeur *Minilisp* correspondante
`(eval x)`	Exécute le code situé dans `x` (typiquement renvoyé par `read`) en appelant `platform_eval`
`(print s)`	Affiche la (liste de) chaîne(s) de caractères `s`
`(mount p h)`	Monte le fichier à l'emplacement `p`, où `h` est une liste de fonctions permettant d'opérer sur le fichier (inutilisé)
`(mmap p)`	Applique l'opérateur `mmap` du système de fichier considéré au fichier à l'emplacement `p`
`(open p)`	Ouvre le fichier à l'emplacement `p`
`(recv f)`	Lit le fichier `f`
`(send f s)`	Écrit la (liste de) chaîne(s) de caractères `s` dans le fichier `f`

TABLE 6 – Liste des *builtins Minilisp* de gestion des fichiers

2 Allocation de la mémoire

2.1 Représentation des valeurs *Minilisp*

Une valeur *Minilisp* est représentée par la `struct Cell` dont on donne la signature :

minilisp.h[81-91]

```
81 typedef struct Cell {
82   union ar {
83     jit_word_t value;
84     void* addr;
85   } ar;
86   union dr {
87     jit_word_t size;
88     void* next;
89   } dr;
90   jit_word_t tag;
91 } Cell;
```

LISTING 7 – Représentation des valeurs

La tête est représentée par le champ `ar` qui est soit une valeur soit un pointeur vers une autre `Cell`. La queue est représentée par le champ `dr` qui est soit une taille soit un pointeur sur la suite de la liste. Enfin, le champ `tag` indique le type de la cellule, qui peut être un entier (`TAG_INT`), une lilste (`TAG_CONS`), un symbole (`TAG_SYM`), une fonction (`TAG_LAMBDA`) ... Le `tag` détermine quels champs des unions seront effectivement utilisés.

2.2 Gestion de la mémoire

La gestion de la mémoire est effectuée par :

— un tableau `cell_heap` qui contient les cellules déjà allouées (qu'elles soient libres ou occupées, l'allocation s'effectuant une fois pour toute lors de l'initialisation du tableau),

— un tableau `byte_heap` qui contient les données non formatées allouées (non implémenté, dans la version courante il s'agit d'un simple `malloc` sans *garbage collection*,

— un tableau `free_list` qui contient les adresses des cellules libres qui ne sont pas en position terminale (au fur et à mesure de l'exécution, il y a des « trous » dans le tableau),

— deux entiers `free_list_avail` et `free_list_consumed` qui sont respectivement la taille du tableau précédent et l'indice de la première cellule non encore réutilisée dans ce même tableau.

Les tailles des différents tableaux sont totalement arbitraires : la limite est de 100 000 cellules au total, et 8 kilooctets de mémoire pour le `byte_heap`.

L'allocation d'une cellule se fait de deux manières différentes : soit une cellule libérée par une précédente itération du *garbage collector*, et en ce cas on utilise celle-ci, soit il n'y en a pas et on cherche à la position courante du `cell_heap` si il reste des cellules non utilisées.

2.3 *Garbage collection*

On dispose d'une fonction `mark_tree` qui parcourt récursivement une cellule (et les cellules vers lesquelles elle pointe) pour les marquer (complexité linéaire en la taille de l'arbre, puisqu'on s'arrête lorsqu'on tombe sur une cellule déjà marquée : ainsi, chaque nœud est visité au plus une fois).

Cette fonction est appelée par `collect_garbage` sur toute l'étendue de la pile (du haut (`stack_end`) vers le bas (`stack_pointer`)). La complexité est encore une fois linéaire en la taille de la mémoire pour la même raison que précédemment.

On va ainsi marquer toutes les cellules atteignables depuis la pile, un système de fichier ouvert ou l'environnement global. On passe alors à la phase de balayage : on parcourt le tas et toute cellule non marquée est donc désignée comme libre.

L'auteur a un problème avec les fonctions (`TAG_LAMBDA`) puisque rien ne pointe vers une clôture anonyme. Il propose pour pallier ce problème de rajouter un pointeur vers les fonctions appelées par une fonction, afin de détecter lorsqu'une fonction ne sera plus jamais appelée de manière certaine.

3 Compilation à la volée

3.1 Fonctions auxiliaires

Parmi les fonctions auxiliaires disponibles, voici les principales :

— `lookup_global_symbol` qui cherche dans l'environnement global l'`env_entry` correspondant au symbole cherché,

— `insert_symbol` et `insert_global_symbol` qui modifient l'environnement pour rajouter une définition d'un symbole,

— `load_int` et `load_cell` produisent du code assembleur pour charger une valeur dans un registre particulier (depuis un autre registre, la pile, le tas, ...),

— `push_frame_regs` et `pop_frame_regs` qui empilent et dépilent certains registres sur la pile (utile pour les appels de fonction, dont le nombre d'arguments passés par registre est fixé à 4),

— `analyze_fn` compte le nombre d'atomes différents dans le corps de la fonction fournie (afin de prévoir la taille de la clôture (`Frame`) qui va être associée à cette fonction),

— `init_compiler` rajoute tous les *builtins* dans l'environnement global.

3.2 La fonction `compile_expr`

Le code s'appuie beaucoup sur le fait que les fonctions écrites en *C* sont accessibles depuis du code assembleur écrit pendant l'exécution du programme, ce qui permet d'utiliser notamment les fonctions d'allocations directement afin de pouvoir faire appel au *garbage collector* non seulement pour le code statique mais aussi pour du code écrit dans le *REPL* par exemple.

On effectue de petites vérifications : par exemple, que les symboles sont bien définis lorsqu'ils sont rencontrés (ou bien dans l'environnement global, ou bien dans la clôture de la fonction). On vérifie également que lorsqu'on a un `TAG_CONS`, la tête est bien une fonction dont la signature concorde avec les arguments fournis. On vérifie également que dans une condition les deux branches

renvoient le même type. On effectue également du typage dynamique lors de l'utilisation d'une fonction comme `get8` afin de vérifier que l'on manipule bien une chaîne de caractères ou des *bytes*.

On adopte une stratégie d'*eager evaluation* pour les appels de fonction : tous les arguments sont d'abord évalués, passés par registres (maximum 4) puis sur la pile et enfin la fonction est appelée. Cependant, il s'agit de bien appliquer une stratégie de *lazy evaluation* pour les définitions de fonctions (on ne souhaite pas que le code soit exécuté lors de la définition de la fonction).

Les fonctions *builtins* faisant appel à des fonctions définies dans du code *C* (par exemple `eval`) sont entourées d'une sauvegarde du pointeur de pile sur le sommet de la pile, ce qui permet de simuler des appels à `call` et `ret`, bien que l'on ne voie pas en quoi c'est nécessaire.

4 Améliorations apportées

4.1 Optimisation du *garbage collecting*

Les appels aux *GC* sont coûteux car il faut alors explorer toutes les références pour savoir si une `Cell` est encore utilisée ou non. Ils sont cependant nécessaires car le programme risque de tomber à court de mémoire. Dans `shell.l`, on remarque qu'après l'exécution et le dessin de chaque *frame*, le *GC* est appelé. Parfois, la mémoire est encore loin d'être pleine et il n'est alors pas nécessaire de faire cet appel. Nous l'avons ainsi remplacé par une nouvelle fonction native (*builtin*) `gc-if-needed` qui appelle le *GC* seulement si la mémoire est trop remplie. De cette manière, il n'y a pas de problème de mémoire et on mesure, lorsque le système est inactif un gain de performances de 50%. Malheureusement, en activité, les gains sont bien moins importants car la mémoire est utilisée intensivement et se remplit vite.

On a obtenu les résultats suivants :

***GC* systématique :** 60 images en 90 unités de temps

***GC* si nécessaire :** 60 images en 60 unités de temps

Implémentation :

alloc.c[186-191]

```
186 Cell* collect_garbage_if_needed(env_t* global_env, void* stack_end, void*
    ↪  stack_pointer) {
187         if(MAX_CELLS - cells_used + free_list_avail - free_list_consumed <
            ↪  MAX_CELLS / 2) {
188                 //printf("run gc (free: %d)\r\n", (MAX_CELLS - cells_used +
                    ↪  free_list_avail - free_list_consumed) * 100 / MAX_CELLS);
189                 collect_garbage(global_env, stack_end, stack_pointer);
190         }
191 }
```

Listing 8 – Appel du *GC* si nécessaire

compiler_new[1638-1646]

```
1638     case BUILTIN_GC_IF_NEEDED: {
1639       push_frame_regs(frame->f);
1640       jit_lea(ARGR0,global_env);
1641       jit_movi(ARGR1,(jit_word_t)frame->stack_end);
1642       jit_movr(ARGR2,RSP);
1643       jit_call3(collect_garbage_if_needed,"collect_garbage_if_needed");
1644       pop_frame_regs(frame->f);
1645       break;
1646     }
```

Listing 9 – Ajout de la primitive

os/shell.l[264-264]

```
264   (gc-if-needed) ; @gc -> gc-if-needed
```

Listing 10 – Appel de la fonction

4.2 Système de tâches

Le système de tâches utilisé par *Interim OS* a l'avantage de pouvoir gérer plusieurs fenêtres à la fois. Cependant, on remarque en pratique qu'il est pourvu de nombreux dysfonctionnements : les `segfaults` sont légion (ils peuvent se produire lors du redimensionnement d'une fenêtre par exemple...), le recouvrement des fenêtres n'est pas bien géré, et enfin celles-ci ne se redessinent pas toujours automatiquement (notamment le logo qui peut être effacé en glissant l'éditeur par-dessus puis en le remettant à sa place initiale).

On a alors effectué les ajouts suivants :

— `desktop-task` pour afficher correctement le logo au fond et du texte,

— `force-draw` qui donne un rendu correct (en provoquant le dessin des fenêtres inactives qui ont put être recouvertes) mais qui est très lent.

os/shell.l[227-234]

```
227 (def max-task-id (+ max-task-id 1))
228
229 (def desktop-task (new task))
230 (sput desktop-task id (+ max-task-id 1))
231 (sput desktop-task name "desktop")
232 (sput desktop-task redrawn 1)
233 (add-task desktop-func desktop-task 0)
234 (def max-task-id (+ max-task-id 1))
```

Listing 11 – Ajout de la tâche de bureau

4.3 Analyse de la lenteur du rendu

Il se trouve que le rendu des fenêtres d'*Interim OS* est très lent. En effet une simple remise à blanc de l'écran par *frame* impacte drastiquement les performances en ralentissant l'exécution de l'ordre de 5 fois. Ceci n'est pas étonnant car toutes les opérations se font pixel par pixel.

D'une part, ce genre de travail sur des pixels n'est pas adapté pour un *CPU* et devrait être exécuté par un *GPU*. D'autre part, il est normalement possible de faire de telles opérations simples et de s'en tirer avec une vitesse d'affichage correcte même sans accélération *GPU*, mais ici pour chaque pixel, il y a un certain coût fixe lié à l'interfaçage entre *Minilisp* et le `Filesystem` représentant l'écran puis le programme *C*.

Ainsi pour résoudre ce problème, il faudrait ajouter des primitives pour remplir directement une zone entière d'une certaine couleur. Si on assimile l'écran à un carré de pixels de coté n ,alors on divise ainsi le coût fixe par un facteur proportionnel à n^2 puisqu'on ne le paye désormais qu'une seule fois par rectangle au lieu d'une fois par pixel.