Projet avancé : étude d'*Interim OS*

Marc Ducret Florentin Guth

9 janvier 2017

Différents niveaux

2 couches

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, . . .).

Fonctionnement général

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*).

Fonctions présentes

On a accès à toutes les primitives *C* qui permettent par exemple de changer la couleur d'un pixel de l'écran, de lancer le *garbage collector*...
Les entrées/sorties du point de vue de l'OS sont toutes représentées par des `Filesystems` (souris, clavier, réseau, disque dur, écran). Ceci permet une certaine abstraction pour la gestion de ceux-ci (par exemple : `(load "/framebuffer/width")`).

Gestion des fenêtres

Le fichier `shell.l` se charge de gérer les fenêtres : il maintient une liste de tâches, chacune représentant une fenêtre.
Il y a une tentative de ne pas tout dessiner à chaque cycle pour ne recalculer uniquement les pixels qui sont modifiés. Le résultat est cependant décevant : souvent certains pixels ne sont pas mis à jour mais devraient l'être. Ainsi, lorsque l'on déplace une fenêtre, l'*OS* ne recalcule que la fenêtre déplacée alors qu'il faudrait également le faire pour les fenêtres recouvertes par celle-ci.

Structure de données

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;
```

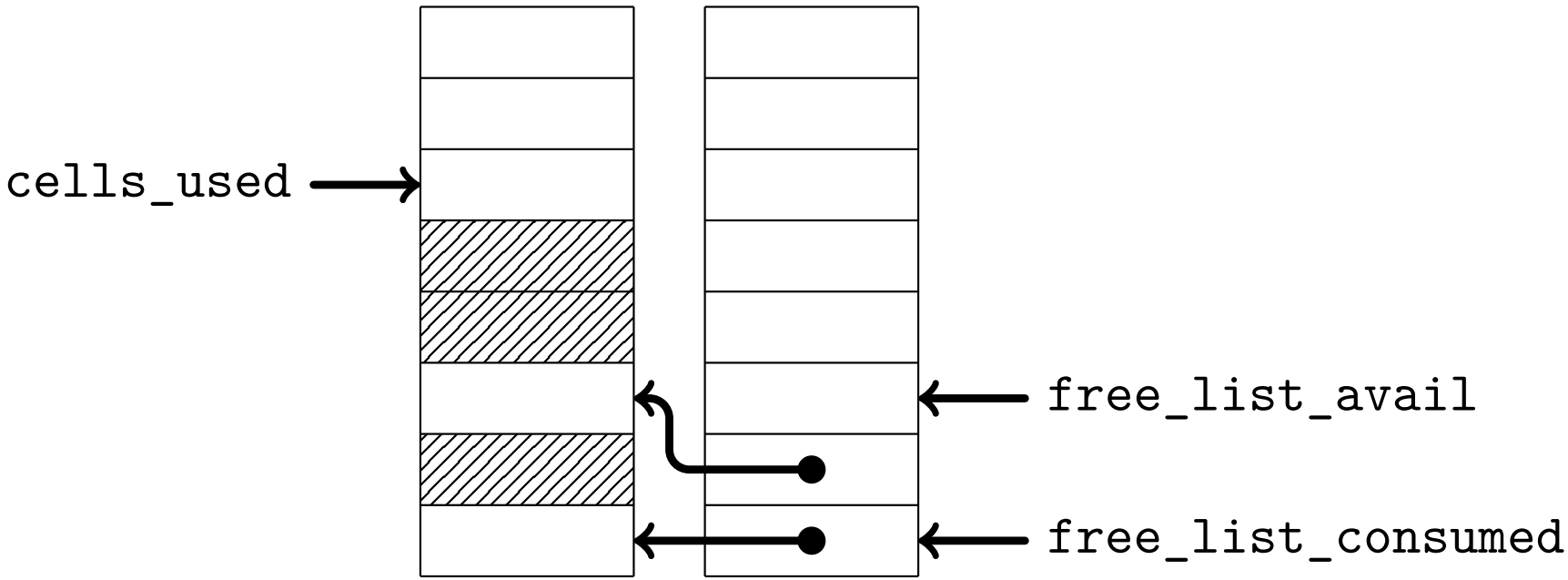
Listing 1 – La `struct Cell`

Avantages

Cette représentation a plusieurs avantages :

- Elle permet d'effectuer facilement (en théorie...) la *garbage collection*, car en fonction du tag on sait si on a affaire à des entiers ou des pointeurs,
- Toute les valeurs ont la même taille (sans compter les cellules pointées par l'un des membres), et il n'y a pas de tableau, ce qui simplifie la gestion des blocs en mémoire.

Schéma de la mémoire



Fonctionnement du GC

Interim OS utilise un *GC mark-and-sweep* classique.

GC mark-and-sweep

Un *GC mark-and-sweep* est un *GC* qui parcourt en profondeur le graphe des `Cells` en marquant les nœuds rencontrés. Dans une deuxième phase, on libère (*sweep*) les nœuds non visités en parcourant l'intégralité du tas.

Les racines sont composées de :

- la pile,
- les fichiers ouverts,
- l'environnement global.

Complexité

L'allocation d'une cellule s'effectue en $O(1)$.
Le marquage se fait en temps $O(|\text{tas}| + |\text{pile}| + |\text{env}| + |\text{fichiers}|)$ et en espace $\Omega(\text{profondeur du tas})$. Il est possible de l'effectuer en espace constant à l'aide de l'algorithme *pointer reversal* qui permet d'encoder la pile de récursion dans le tas lui-même.
Le balayage se fait en temps $O(|\text{tas}|)$, et en espace en plus des deux tableaux précédents $O(1)$.

Mélange C-assembleur

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.

Remarques notables

On effectue de petites vérifications : définitions des symboles, typage statique et typage dynamique (lors de l'utilisation d'une fonction comme `get8`).
On adopte une stratégie d'*eager evaluation* pour les appels de fonction (passage par valeur) : tous les arguments sont d'abord évalués, passés par registres (maximum 4) puis sur la pile et enfin la fonction est appelée.
Lorsqu'on compile des définitions de fonctions, on précalcule l'espace de pile nécessaire à chaque fonction pour l'allouer en une seule fois lors de l'appel.

Appels que lorsque nécessaire

Nous avons remplacé les appels au *GC* par une nouvelle *builtin* `gc-if-needed` qui appelle le *GC* seulement si la mémoire est trop remplie.

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 2 – 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 3 – Ajout de la primitive

Tâche du bureau

Pour pallier aux problèmes précédemment cités, on a 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 pu être recouvertes) mais qui est très lent.

Implémentation

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 4 – Ajout de la tâche de bureau

Raisons

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.
Il y a deux raisons à cela :

- les opérations graphiques devraient être gérées par le *GPU*,
- 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*.

Solution proposée

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.

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 – Liste des fichiers *C*

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 – Liste des fichiers *Minilisp*

Arithmétique et logique

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 – Liste des *builtins Minilisp* arithmético-logiques

Contrôle

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 – Liste des *builtins Minilisp* de contrôle

Mémoire

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

Fichiers

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 – Liste des *builtins Minilisp* de gestion des fichiers