Interopérabilité : Notes de cours

LU3IN032 : Programmation comparée

Basile Pesin

19 mars 2021

Table des matières

Résumé

De tout temps, l'humanité à voulu rendre compatibles différents langages de programmation.

On appelle "intéropérabilité" la capacité de deux langages à intéragir et communiquer au sein d'un même système. Rendre deux langages ainsi compatibles présente plusieurs intérêts pour les développeurs : premièrement, des langages différents sont adaptés pour résoudre des problèmes différents, selon leurs caractéristiques (cf ce cours !). Par exemple, une application écrite dans un langage de haut niveau (OCaml, Java, Python), pourrait bénéficier des performances d'un langage bas-niveau (C), ou de la possibilité d'accéder directement au matériel. Par ailleurs, on pourrait vouloir profiter des bibliothèques de fonctionnalités déjà écrites dans un autre langage, et mélanger les styles de programmations propres à différents langages.

1 Langages de haut et bas niveau : OCaml et C

Dans cette première partie, on verra comment un langage de haut niveau peut accéder à un langage de plus bas niveau. En l'occurrence, on étudiera principalement les interactions du langage OCaml avec le langage C. On verra aussi que ce fonctionnement est très proche de celui de Java.

1.1 Machine virtuelle et Foreign Function Interface

L'exécution du bytecode produit par le compilateur OCaml passe par l'utilisation d'une machine virtuelle (ZAM). Il se trouve que cette machine est écrite dans le langage C. Il est donc a priori facile pour cette machine d'appeller du code C défini par l'utilisateur ([2], [1]).

Les instructions de la famille `C_CALL` prennent comme argument un pointeur de fonction indiquant la fonction C à exécuter.

L'utilisateur peut déclarer l'utilisation d'une fonction C grâce au mot clé `external`. Il précise le type de la fonction, ainsi que son nom dans le code C (voir 1).

```
type c_list

external empty_c_list : unit -> c_list = "empty_c_list"
external cons_c_list : int -> c_list -> c_list = "cons_c_list"
external print_c_list : c_list -> unit = "print_c_list"

let _ =
  let l = empty_c_list () in
  let l1 = cons_c_list 42 (cons_c_list 21 l) in
  print_c_list l;
  print_c_list l1
```

Listing 1 – Utilisation d'une liste chaînée implémentée en C

Les fonctions C doivent bien sûr être définies dans un fichier `.c`. Ici le fichier `clist.c` est détaillé dans 2.

```
#define CAML_NAME_SPACE
#include <caml/mlvalues.h>

typedef struct _List {
  int data;
  struct _List *next;
} List;

CAMLprim value empty_c_list(value u) { [...] }

CAMLprim value cons_c_list(value iv, value lv) { [...] }

CAMLprim value print_c_list(value lv) { [...] }
```

Listing 2 – Fonctions de gestion de liste chainée en C

L'exécutable `clist.byte` peut être compilé avec la série de commande suivantes :

```
# Compile clist.c vers une bibliotheque clist.a
gcc -o clist.o clist.c -I~/.opam/default/lib/ocaml
ar rcs clist.a clist.o
# Compile clist.ml vers un executable, en utilisant la biblioth[U+FFFD]que clist.a
ocamlc -c -o clist.cmo clist.ml
ocamlc -o clist.byte -custom clist.cmo -cclib -L. -cclib -lclist
```

En Java :

Tout comme la ZAM, la JVM (Java Virtual Machine) est écrite en C. On peut donc également définir une bibliothère C à intégrer en utilisant la JNI (Java Native Interface).
L'utilisateur peut déclarer des méthodes `native`. L'outil `javah` se charge de générer automatiquement les headers correspondant aux méthodes déclarées.

1.2 Représentation des valeurs

Les valeurs OCaml sont encapsulées dans le type `value`. Le fichier `caml/mlvalues.h` fourni des accesseurs / constructeurs permettant de les manipuler. Une valeur peut être de deux formes : entiers (non alloués) et pointeur vers un "bloc" alloué dans le tas. Les blocs sont tagués, ce qui permet de reconnaître leur contenu, et contiennent un certain nombre de champs.

Il faut vérifier dynamiquement la forme de la valeur ! On perd la sécurité du système de types.

1.2.1 Types de bases

Type OCaml	Type C	Tag	value -> C	C -> value
`int`	`int`	NA	`Int_val`	`Val_int`
`bool`	`int`	NA	`Bool_val`	`Val_bool`
`float`	`double`	`Double_tag`	`Double_val`	`Store_double_val`
`string`	`char *` ou `const char*`	`String_tag`	`String_val`	`caml_alloc_initialized_string`

1.2.2 n-uplets, enregistrements

Les n-uplets et enregistrements (types produits en OCaml) sont simplement stockés comme des blocs avec un tag 0. Leurs champs peuvent être lus grâce à la macro `Field(b, n)`, et modifiés par la fonction `Store_field(b, n, v)`. Il est intéressant de noter que tous les champs des n-uplets et enregistrements sont modifiables en place, même si non-mutables. Par exemple dans :

```
type rec2 = {
  a1 : string;
  a2 : int; (* Pas mutable ! *)
  a3 : bool;
}

external update_record : rec2 -> unit = "update_record"

let _ =
  let r1 = {a1 = "coucou";
            a2 = 41;
            a3 = false} in
  update_record r1;
  print_int r1.a2;
```

Il est tout à fait possible que la valeur affichée soit autre que `41`, puisqu'elle peut être modifiée par `update_record`. On voit qu'on perds ici la garantie d'immutabilité proposée par le système de types.

1.2.3 Types sommes, ADTs

Les types sommes sont eux aussi encodés comme des valeurs. Les constructeurs constants (sans paramètre) sont simplement encodés comme des entiers. Les constructeurs prenant des paramètres sont eux encodés comme des blocs, avec comme taille le nombre de paramètres. L'ordre des constructeurs dans la définition du type induit la numérotation des tags. La numérotation des constructeurs constants et non-constants est indépendante (voir 3).

```
type t =
  | A             (* Val_int(0) *)
  | B of string   (* bloc, tag 0, taille 1 *)
  | C             (* Val_int(1) *)
  | D of bool     (* bloc, tag 1, taille 1 *)
  | E of t * t    (* bloc, tag 2, taille 2 *)
```

Listing 3 – Exemple d'encodage d'un type somme

1.2.4 Abstract et Custom types

Enfin, on peut souhaiter représenter des types "abstraits", c'est-à-dire dont on ne connaît pas la représentation dans le code OCaml. C'est utile en particulier quand on veut accéder à une bibliothèque existante, dans laquelle des structures de données sont déjà définies. Dans l'exemple donné en 2, on peut représenter notre liste comme un type abstrait. Le tag à utiliser est alors `Abstract_tag`.

```
CAMLprim value empty_c_list(value unit) {
  CAMLparam1(unit);
  CAMLlocal1(v);

  // We create an empty list
  List *l = NULL;

  // That we pack as a caml value
  v = caml_alloc(1, Abstract_tag);
  List_val(v) = l;
  CAMLreturn(v);
}
```

Listing 4 – Construction d'une valeur de type abstrait

Parfois, on peut vouloir donner un peu plus d'opérations sur ces blocs "abstraits". Pour cela, on peut les définir comme "custom", identifiés par le `Custom_tag`.

Lors de l'allocation avec `caml_alloc_custom(ops, size, mem, max)`, on passe en premier paramètre une structure `custom_operations` :

```
struct custom_operations {
  char const *identifier;
  void (*finalize)(value v);
  int (*compare)(value v1, value v2);
  intnat (*hash)(value v);
  void (*serialize)(value v, uintnat * bsize_32, uintnat * bsize_64);
  uintnat (*deserialize)(void * dst);
  int (*compare_ext)(value v1, value v2);
  const struct custom_fixed_length* fixed_length; };
```

La fonction `finalize` permet de traiter la libération de la mémoire du bloc.

Les fonctions `compare` et `compare_ext` permettent de définir la comparaison entre deux valeurs, et donc d'utiliser les opérateurs de comparaison polymorphe d'OCaml. `compare_ext` est utilisée si l'un des paramètres est un entier et l'autre est un bloc custom, ce qui est un cas assez rare.

La fonction `hash` permet de définir le hashing d'une valeur, comme son nom l'indique.

Les autres champs et fonctions sont utilisées pour sérialiser ou désérialiser une valeur.

En Java :

Les valeurs java sont séparées en deux catégories : les valeurs primitives (valeurs numériques telles que `int`, `float`, `boolean`, ...) et les objets. Pour les premiers, ils apparaissent dans la signature de la fonction C comme `jint`, `jfloat`, `jboolean`, ... et peuvent être utilisées directement. C'est aussi le cas de certains objets de classes particulières (`String` par exemple qui apparait comme `jstring`), pour lesquelles la JNI propose des fonctions prédéfinies.
Les autres objets apparaissent comme `jobject`. Des fonctions de la JNI permettent l'accès et l'appel de méthodes sur ces objets, en les cherchant par nom. Bien sûr, la présence d'une méthode du bon nom n'est vérifiée que dynamiquement.

1.3 Gestion de la mémoire

Comme vous l'avez étudié dans un cours précédent, les langages de haut-niveau tels qu'OCaml ou Java bénéficient souvent d'un mécanisme de gestion de la mémoire automatique (GC, pour garbage collector ou glaneur de cellules). Ce n'est pas le cas du langage C, où la libération de la mémoire est à la charge du programmeur. Pour que ces deux modèles coopèrent, il faut que le code C et sa gestion manuelle donne beaucoup d'informations aux GC, pour que celui-ci puisse fonctionner correctement.

Les variables présentes sur le tas et qui donc doivent être manipulées par le GC sont celles ayant le type `value`, comme on l'a vu plus haut. En particulier, les blocs doivent être effectivement alloués sur le tas afin de pouvoir être collectés par le GC plus tard. Pour ce faire, on utilise la fonction `caml_alloc(size, tag)` qui alloue un bloc de taille `size`, avec le `tag` donné. Cependant, on s'aperçoit tout de suite d'une difficulté : si on peut allouer dans le tas, alors il est tout à fait possible que le GC se déclenche pendant l'exécution d'une fonction externe. Il va donc falloir s'assurer que le GC dispose des bonnes informations, même au milieu d'une fonction externe, afin de ne pas libérer de la mémoire utilisée.

Une fonction C destinée à être appellée par la FFI doit toujours commencer par l'une des macros de la famille `CAMLparam` (`CAMLparam0()`, `CAMLparam1(x)`, `CAMLparam2(x, y)`,...). Cette macro alloue un cadre sur la pile d'appels de fonctions, et y place les variables paramètres de la fonction. Ainsi, ces pointeurs seront pris en compte comme racines si le GC se déclenche pendant l'exécution de la fonction, et les blocs associés ne seront pas libérés. De la même manière, les variables locales à la fonction doivent aussi être enregistrées dans le cadre. Cela peut être fait via la famille de macros `CAMLlocal` (`CAMLlocal1(x)`, ...). On utilise ces macros à la place de la déclaration d'une variable locale (par exemple `value v;` est remplacée par `CAMLlocal1(v)`).

Bien sûr, aucune de ces règles ne sont vérifiées à la compilation du programme, ce qui induit encore un risque d'erreur de la part du programmeur. Il faut donc être particulièrement délicat.

```
CAMLprim value cons_c_list(value iv, value lv) {
  CAMLparam2(iv, lv);
  CAMLlocal1(nv);
```

```
  // Get the C values back
  List *l = List_val(lv);
  int i = Int_val(iv);

  List *nl = malloc(sizeof(List));
  nl->data = i;
  nl->next = l;

  // Return the new list, packed as a caml value
  nv = caml_alloc(1, Abstract_tag);
  List_val(nv) = nl;
  CAMLreturn(nv);
}
```

Listing 5 – Gestion de la mémoire dans la fonction `cons_c_list`

Dans la figure 5, on montre un exemple de l'utilisation de toutes ces macros pour bien définir la fonction qui ajoute un entier en tête d'une liste chainée en C.

En Java :

La JVM dispose aussi d'un glaneur de cellule, et donc les mêmes questions de gestion mémoire se posent. Lors de l'accès à des Objets par la JNI, on n'utilise pas directement les addresses réelles des objets sur le tas : c'est parce que le GC de Java, qui pourrait se déclencher pendant l'exécution de la fonction native, et éventuellement déplacer certains objets dans le tas lors de la phase de compaction. L'utilisateur manipule donc des références plutôt que des pointeurs directs.
Plus de détails sont disponibles en [3].

1.4 Exceptions

Il est également possible de déclencher des exceptions depuis le code C. Pour les exceptions "classiques" `Failure` et `Invalid_argument`, les fonctions `caml_failwith(s)` et `caml_invalid_argument(s)` sont fournies.

Utiliser une exception définie par l'utilisateur est plus complexe. L'utilisateur doit enregistrer l'exception sous un nom, qui permettra de l'identifier dans le code C.

Par exemple, on enregistre l'exception `Error` :

```
exception Error of string
let _ = Callback.register_exception "my_error" (Error "dummy")
```

On note que la chaîne passée au constructeur `Error` ne sera pas prise en compte quand l'exception sera levée, et ne sert qu'à produire une expression du bon type. On peut alors lever l'exception depuis le code C. Pour cela, on récupère la valeur identifiée grâce à `caml_named_value`. Plusieurs fonctions permettent alors de lever l'exception :

- `caml_raise_constant(e)` pour une exception sans paramètre
- `caml_raise_with_arg(e, v)` pour une exception avec un paramètre
- `caml_raise_with_args(e, n, v)` pour une exception à `n` arguments (`v` est un tableau de `value` de taille `n`)
- `caml_raise_with_string(e, msg)` pour une exception avec comme seul paramètre une chaîne de caractères.

Ici on peut utiliser cette dernière fonction :

```
void raise_error(char * msg) {
  caml_raise_with_string(*caml_named_value("my_error"), msg);
}
```

1.5 Callbacks

Jusqu'ici, on a vu comment écrire du code C et l'appeler depuis OCaml. Cependant, il est aussi possible de rappeler du code OCaml depuis une fonction C. En effet, OCaml permettant d'écrire des fonctions d'ordre supérieur (prenant des fonctions en paramètre), on peut supposer une fonction avec la signature suivante : `apply : ('a -> 'b) -> 'a -> 'b` dont le rôle est simplement d'appliquer son premier paramêtre à son deuxième, est de renvoyer le résultat. Comment écrire cette fonction en C ? Très simplement en utilisant le mécanisme de "callbacks". On définit la fonction `apply` en C. Le premier de ses paramètres est la fermeture (closure) passée à la fonction. Cette fermeture peut être appelée grâce à la fonction `caml_callback`. Celle ci lance tout simplement une nouvelle instance de l'interpréteur OCaml, sur le code de la fermeture.

```
CAMLprim value apply(value closure, value arg) {
  CAMLparam2(closure, arg);

  CAMLreturn(caml_callback(closure, arg));
}
```

```
external apply : ('a -> 'b) -> 'a -> 'b = "apply"

let _ =
  print_int (apply (fun x -> x * 2) 21)
```

En Java :

Si les versions récentes de Java proposent des fonctions d'ordre supérieur, celles-ci sont en fait compilées vers un appel à une fonction statique appartenant à une interface attendue. Le mécanisme d'appel à une telle fonction depuis la JNI est donc identique à l'appel "classique" d'une méthode de classe depuis la JNI.

2 Performances : Python vs Cython

Le langage Python est en général interprété. Cependant, l'usage d'un interpréteur condamne toute optimisation effectuée durant la compilation, et limite donc les performances du programme.

Le compilateur Cython permet, en compilant le langage Python vers C, de profiter à nouveaux de telles optimisations. Par ailleurs, le compilateur Cython permet également de faire directement appel à des fonctions externes C, similairement à l'approche d'OCaml, mais nous ne détaillerons pas cette possibilité ici.

On présente en 6 une implémentation naïve de la méthode de Newton (qui permet d'approximer la racine du fonction). Une fois le fichier contenant cette définition chargé par l'interpréteur Python, entrer `newton(sq1, sq1der, 1)` permet de calculer une approximation de la racine de 2.

```
def sq1(x):
    return x**2 - 2
```

```
def sq1der(x):
    return 2*x

def newton1(f, der, init):
    epsilon = 0.001
    x = init

    while True:
        x1 = x - f(x)/der(x)

        if abs(x - x1) < epsilon:
            return x1
        x = x1
```

Listing 6 – Méthode de Newton, en Python

Cependant, ce programme peut également être compilé via Cython. Le processus de compilation produit alors une bibliothèque (fichier `.so`) qui peut être chargée dans l'interpréteur Python (via la directive `import`). En expérimentant, on se rend compte que la version compilée du programme est environ 2 fois plus rapide que la version interprétée (voir la table 1).

On peut faire mieux ! Jusqu'ici on a manipulé du code Python "classique". En 7 on présente la version Cython du programme. Il s'agit presque du même programme, mais on y a ajouté des annotations de types. Celles-ci sont vérifiées statiquement à la compilation, et permettent d'éviter de nombreux tests dynamiques de type lors de l'exécution. Le temps d'exécution du programme est divisé par 15, par rapport à la version non typée.

```
cdef float sq2(float x):
    return x**2 - 2

cdef float sq2der(float x):
    return 2*x

ctypedef float (*ffunc)(float)

cdef newton2(ffunc f, ffunc der, float init):
    cdef float epsilon, x, x1
    epsilon = 0.001
    x = init

    while True:
        x1 = x - f(x)/der(x)

        if abs(x - x1) < epsilon:
            return x1
        x = x1
```

Listing 7 – Méthode de Newton, en Cython

Les gains de performance peuvent être encore plus significatifs pour les programmes manipulant des tableaux.

On présente en 8 un petit exemple de tri par insertion en Python. Le temps de calcul est en O(n^2) par rapport à la longeur de la liste (ce n'est pas un algorithme de tri très efficace, mais il s'écrit très facilement dans un style impératif).

```
def insert_sort(l):
    n = len(l)
    for i in range(1, n):
        x = l[i]

        while l[i-1] > x and i > 0:
            l[i], l[i-1] = l[i-1], l[i]
            i = i - 1
    return l
```

Listing 8 – Tri par insertion, en Python

La version typée du programme est donnée en 9. Celle-ci utilise des "Typed MemoryViews" qui permettent un accès efficace aux zones mémoires dans lesquelles les tableaux sont stockés. Une difficulté est qu'on ne peut pas utiliser un tel accès avec les listes primitives de python. A la place, on doit construire un tableau de taille fixe (ici avec l'aide de `numpy`). L'accès aux MemoryViews est très rapide, est le temps d'exécution pour l'algorithme de tri par insertion est divisé par 30. On peut faire encore mieux en ajoutant la directive `@cython.boundscheck(False)` avant la déclaration de la fonction. Celle-ci permet de désactiver la vérification dynamique des bornes lors de l'indexage d'un tableau. Si on essaie d'accéder à un élément en dehors d'un tableau, l'erreur `IndexError` ne sera pas lancée : on ne pourra donc pas la rattraper avec un récupérateur d'exception par exemple. A la place, on aura un comportement indéterminé ou un crash du programme (segmentation fault).

```
@cython.boundscheck(False)
def insert_sort2(int[:] l):
    cdef int n, i, x

    n = l.shape[0]
    for i in range(1, n):
        x = l[i]

        while l[i-1] > x and i > 0:
            l[i], l[i-1] = l[i-1], l[i]
            i = i - 1
    return l

l = [random.randint(0, 200) for _ in range(n)]
cdef int [:] lv = np.array(l, dtype=np.dtype("i"))
insert_sort2(lv)
```

Listing 9 – Tri par insertion, en Cython

On synthétise dans la table 1 les performances des deux programmes vus ci-dessus, ainsi que d'une implémentation du crible d'Erathostène, et de la multiplication (naïve) de matrices. On constate que les gains de performance dans la version typée du programme sont toujours significatives, en particulier dans les cas où l'on manipule des tableaux (multiplication de matrices par exemple).

3 Langages à cible commune

Plus haut, on a décrit la façon dont un langage de haut-niveau intéragit avec le langage de bas niveau permettant son implémentation. Il existe une autre forme d'interopérabilité : plusieurs

Table 1 – Performances comparées de Python et Cython

Programme	Interpreté	Compilé, non typé	Compilé, typé	Compilé, typé + `boundscheck(False)`
Newton	1.13s	0.69s	0.04s	0.04s
InsertSort	4.07s	1.98s	0.07s	0.05s
Erathostene	1.73s	1.23s	0.85s	0.82s
MatMult	6.52s	2.44s	0.11s	0.05s

langages de haut-niveau peuvent également intéragir, pour peu qu'ils partagent un même environnement d'exécution.

3.1 Une plate-forme pour les gouverner tous : la JVM

Le slogan "Write once, run anywhere" créé par Sun Microsystems pour promouvoir le langage Java illustre une idée importante : une machine virtuelle permet de factoriser les efforts des développeurs lors du portage vers différentes architectures, systèmes d'exploitations, etc... En théorie, un programme Java peut être exécuté sur n'importe quel environnement, pour peu que la JVM y ait été portée. Cette machine virtuelle est l'une des clés de la réussite industrielle du langage Java. Cette réussite est d'ailleurs quantifiée par un nouveau slogan (que vous avez sûrement déjà lu) : "3 Billion Devices Run Java", qui fait état de la très large installation de la JVM.

Cette base d'installation conduit à la réalisation suivante : d'autres langages ciblant la même machine virtuelle bénéficient alors de la même base d'installations. De nombreux langages autres langages peuvent donc tirer partie de la JVM. Certains sont écrits spécifiquement avec la JVM en tête (Clojure, Scala, Kotlin...), mais des implémentations utilisant la JVM ont aussi été développées pour des langages pré-existants (Jython pour Python, OCaml-Java pour OCaml...).

L'utilisation d'une plateforme commune impose des contraintes sur le design ou sur la compilation des langages, mais elle rend aussi beaucoup plus simple l'interopérabilité entre langages. Plus bas, on voit comment le langage Clojure interagit avec Java.

3.1.1 Interactions Clojure-Java

Il est très facile de manipuler des instances de classes Java : un nouvel objet peut être créé grâce au mot-clé `new`.

```
(def p (new java.awt.Point 1 2))
(.translate p 2 3)
(.getX p) ;; 3.0
(.-x p)   ;; 3
```

Listing 10 – Instantiation et manipulation d'un objet

Pour simplifier le chaînage des appels des méthodes, le langage propose l'utilisation de la macro `doto`.

```
(def list (new java.util.ArrayList))
(doto list (.add 41) (.add 42))
```

Listing 11 – Macro `doto`

3.1.2 Démo live : Yaw

A découvrir sur `https://github.com/yaw-central/`.

3.2 JavaScript : l'assembleur du Web

Nous avons vu dans précédemment qu'il était facile de faire communiquer plusieurs langages exécutés par une machine virtuelle commune. Il se trouve que tous les navigateurs web modernes embarquent une machine virtuelle pour l'exécution du langage JavaScript.

Si cette machine était d'abord destinée spécifiquement à l'exécution de programmes écrits en JavaScript, de nombreux langages compilant vers JavaScript ont depuis vus le jour (Typescript, Clojurescript...).

Pour pouvoir exécuter des programmes existants dans le navigateur, des compilateurs de langages existants vers JavaScript sont aussi disponibles. On verra ici l'exemple de Js_of_ocaml ([4]). Ce compilateur accepte en entrée le bytecode d'un programme OCaml. L'inconvénient de ce procédé est que le code JavaScript produit en sortie du compilateur n'est pas lisible. L'avantage est que, le bytecode OCaml étant assez stable (il est assez rare qu'une instruction y soit ajoutée ou retirée), Js_of_ocaml peut donc aisément rester compatible avec les versions récentes d'OCaml. Par ailleurs, Js_of_ocaml peut également bénéficier des optimisations déjà opérées par le compilateur OCaml.

3.2.1 Utilisation de base

Compiler un programme avec Js_of_ocaml est extrêmement simple : comme présenté en figure 13, on commence par exécuter le compilateur `ocamlc` sur un fichier source, puis on exécute `js_of_ocaml` sur le bytecode résultant. On peut alors exécuter le programme `hello.js` produit, par exemple en utilisant Node.js.

```
let _ =
  print_endline "Hello World"
```

Listing 12 – `hello.ml`

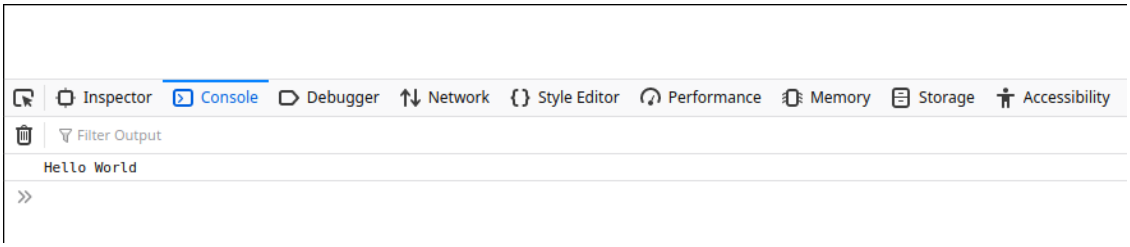
```
ocamlc -o hello.byte hello.ml
js_of_ocaml hello.byte
node hello.js # affiche "Hello World"
```

Listing 13 – Compilation avec Js_of_ocaml

Bien sûr, une utilisation plus intéressante est d'intégrer le script produit dans une page web. Pour cela, on utilise simplement la balise `<script src="hello.js"></script>`. Au chargement de la page, le message "Hello World" s'affiche effectivement dans la console développeurs.

```
<html>
    <head>
        <title>Hello World</title>
        <meta charset="UTF-8" />
    </head>
    <body>
        <script src="hello.js"></script>
    </body>
</html>
```

Listing 14 – Page `hello.html` chargeant `hello.js`



3.2.2 Manipulation d'objets JS

Pour intéragir avec des programmes JavaScript existants, on souhaite pouvoir manipuler les types de données de JavaScript, et en particulier ses objets.

Si les objets JavaScript ne sont pas statiquement typés (puisque le langage ne l'est pas), on veut en revanche pouvoir les manipuler avec un typage fort dans OCaml. Js_of_ocaml introduit le type `'a Js.t`, le type d'un objet JavaScript. On donne ci-dessous un exemple d'objet JavaScript déclaré en OCaml. On note que la fonction `Js.string` permet de convertir une chaîne de caractères OCaml en chaîne JavaScript.

```
let albert = object%js
  val surname = Js.string "Albert"
  val name = Js.string "Einstein"
  val mutable age = 42
end

albert : < age : int Js.prop;
           name : Js.js_string Js.t Js.readonly_prop;
           surname : Js.js_string Js.t Js.readonly_prop > Js.t
```

Listing 15 – Objet `albert` et son type

Il est aussi possible de déclarer directement un type d'objets. Attention ! Quand on déclare un type d'objet, tous les champs doivent être déclarés comme `method` et pas comme `val`. Les déclarations de types permettent de différentier méthodes et champs. Par exemple, la méthode `inc_age` renvoie un `unit Js.meth`, tandis que les autres champs sont déclarés comme `Js.readonly_prop` ou `Js.prop`.

```
class type person = object
  method surname : Js.js_string Js.t Js.readonly_prop
  method name : Js.js_string Js.t Js.readonly_prop
  method age : int Js.prop
  method inc_age : unit -> unit Js.meth
end

let albert : person Js.t = object%js(self)
  val surname = Js.string "Albert"
  val name = Js.string "Einstein"
  val mutable age = 42
  method inc_age () =
    self##.age := self##.age + 1
end

let _ =
  print_int albert##.age; (* 42 *)
```

```
albert##inc_age ();
print_int albert##.age; (* 43 *)
```

Listing 16 – Type et manipulation d'objet

Js_of_ocaml introduit quelques notations spécifiques pour manipuler les objets. La notation `object%js` permet de construire un objet JavaScript, en utilisant la syntaxe OCaml habituelle. On peut accéder aux attributs d'un objet par la notation `##.`, et appeller les méthodes via `##`.

Il est également possible de créer des objets JavaScript en utilisant des constructeurs existants. Les constructeurs prennent un type spécial, de la forme `('a -> 'b Js.t) Js.constr`, où `'b` est le type de l'objet crée. Les constructeurs peuvent être utilisés grâce à la notation `new%js` ; par exemple, le code ci-dessous initialise un tableau, via le constructeur `Js.array_empty` fourni dans la bibliothèque de Js_of_ocaml. Le programme place ensuite `42` et `22` dans les deux premières cases du tableau. On remarque que le système de types d'OCaml infère alors que `arr` à le type `int Js.js_array Js.t`, et nous empêche donc statiquement de mettre une chaîne de caractères dans le tableau.

```
let _ =
  let arr = new%js Js.array_empty in
  Js.array_set arr 0 42;
  (* Js.array_set arr 1 "coucou"; serait détecté comme malvenu *)
  Js.array_set arr 1 22
```

3.2.3 Interaction avec le DOM

Une des principales raisons d'utiliser JavaScript est de pouvoir intéragir avec directement avec la page Web. Voyons comment écrire un programme OCaml effectuant quelques manipulations simples.

On veut écrire un programme très simple : une page Web munie de deux boutons permettant d'incrémenter et de décrémenter un compteur. Le code HTML de cette page est le suivant :

```
<html>
    <head>
        <title>Compteur</title>
        <meta charset="UTF-8" />
    </head>
    <body>
        <script src="script.js"></script>
        <form action="">
            <input id="decr" type="button" value="décrémenter" />
            <input id="incr" type="button" value="incrémenter" />
        </form>
        <p>Compteur : <b id="counter">0</b></p>
    </body>
</html>
```

Listing 17 – `counter.hmtl`

Dans la figure 18, on présente le programme chargé d'intéragir avec la page Web. Détaillons le ligne par ligne.

Les lignes 1 à 3 permettent d'ouvrir les modules de la bibliothèque de `Js_of_ocaml`, qui contient les fonctionnalités de manipulation du DOM. En particulier, `Js_of_ocaml_tyxml` offre des fonctions simples permettant de créer des éléments HTML à ajouter à la page web.

Les lignes 5 et 7 définissent des racourçis pour des fonctions dont nous auront besoin. `by_id` permet de trouver un élément de la page web par son identifiant, et `of_node` permet de fabriquer un élément de page web depuis la représentation interne de la bibliothèque `Tyxml_js`.

La fonction `replace_child` est un peu plus complexe : elle permet de remplacer le premier enfant d'un élément HTML par un autre élément passé en paramètre. Obtenir "le premier enfant" peut bien sûr échouer si le parent n'a pas d'enfants, auquel cas on lève une exception `Invalid_argument "replace_child"`.

La ligne 14 crée une référence (mutable) vers le compteur lui même, qu'on manipulera donc de manière impérative.

La fonction `update_counter` (lignes 16 à 19) est la fonction qui sera appelé lors de l'appui sur l'un des boutons. Elle met à jour la valeur du compteur, puis met à jour l'élément d'identifiant `"counter"` en y affichant la valeur actuelle du compteur. Pour cela on fabrique un nouvel élément HTML contenant simplement du texte grâce à la fonction `T.txt`. La fonction prend en argument un évènement (dont on ignore le contenu) et doit renvoyer `Js._true` pour réarmer l'évènement.

Dans la fonction `init`, en particuliers aux lignes 22 et 23, on déclare que la fonction `update_counter` sera appelé quand l'utilisateur clique sur l'un des boutons, en définissant leurs attributs `onclick`. Cette fonction elle même sera appelée dès que la page aura fini de charger, comme déclaré ligne 27.

```
1  open Js_of_ocaml
2  open Js_of_ocaml_tyxml
3  module T = Tyxml_js.Html5
4
5  let by_id s = Dom_html.getElementById s
6
7  let of_node = Tyxml_js.To_dom.of_node
8
9  let replace_child parent child =
10   let oldchild = Js.Opt.get (parent##.childNodes##item 0)
11       (fun _ -> invalid_arg "replace_child") in
12   ignore (parent##replaceChild child oldchild)
13
14 let counter = ref 0
15
16 let update_counter diff (_ : #Dom_html.event Js.t) =
17   counter := !counter + diff;
18   replace_child (by_id "counter") (of_node T.(txt (string_of_int !counter)));
19   Js._true
20
21 let init (_ : #Dom_html.event Js.t) =
22   (by_id "incr")##.onclick := Dom_html.handler (update_counter 1);
23   (by_id "decr")##.onclick := Dom_html.handler (update_counter (-1));
24   Js._true
25
26 let _ =
27   Dom_html.window##.onload := Dom_html.handler init
```

Listing 18 – `counter.ml`

Vous pouvez tester ce programme très simple sur le web.

Références

[1] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system : Documentation and user's manual - Interfacing C with OCaml. August 2020.

[2] Pascal Manoury, Chailloux Emmanuel, and Bruno Pagano. Développement d'applications avec Objective Caml, Chapitre 12. 01 2000.

[3] IBM. The Java Native Interface - The JNI and the Garbage Collector.

[4] Jérôme Vouillon and Vincent Balat. From Bytecode to JavaScript : the $\mathrm{Js}_o f_o caml Compiler. Software : Practice and Experience, 44, 082014$.