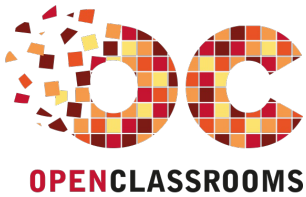
Introduction au dialogue entre OCaml et le C

Par Cacophrene



www.openclassrooms.com

Dernière mise à jour le 15/11/2009

Sommaire

Sommaire 2
Introduction au dialogue entre OCaml et le C 3
Des macros et de la discipline 3
Des macros 3
De la discipline 4
Hello world! 4
La macro CAMLprim 5
La macro CAMLparamN 5
La macro CAMLreturn 5
Compilation 5
Les types de base 5
Les booléens 5
Les entiers 6
Au pays des allocations 7
Les nombres à virgule flottante 7
Les chaînes de caractères 8
Vivre en harmonie avec le ramasse-miettes 9
Interactions avec le ramasse-miettes 9
Variables locales de type value 9
Subtilités 9
Que j'aime à faire apprendre ce nombre aux sages... 9
N-uplets et listes 10
Les n-uplets 10
Les listes 11
Où l'on déshabille un chameau 13
Des avantages et des inconvénients 13
OCaml vu de l'intérieur 13
Le fantasme de la fonction print polymorphe 13
Erreurs fréquentes 13
Des calculs erronés 13
Mise en échec du typage 14
Autres erreurs possibles 15
Remerciements 15
Références bibliographiques 15
Partager 15

Introduction au dialogue entre OCaml et le C

Par  Cacophrene

Mise à jour : 15/11/2009
Difficulté : Intermédiaire

Bonjour à tous !

Il n'est pas rare que l'on souhaite faire dialoguer OCaml avec le C. C'est particulièrement vrai quand :

- On veut utiliser une bibliothèque écrite en C (par exemple GTK+) ou dialoguer avec un langage tiers par l'intermédiaire du C.
- On a identifié les goulots d'étranglement (*bottleneck*) d'un code OCaml et on voudrait gagner en performance en les réécrivant en C.
- On a plus d'imagination que moi pour trouver des exemples d'application. 

Bien entendu, le dialogue entre deux langages n'est pas quelque chose d'anodin. Je vous propose ici une introduction orientée vers la pratique et destinée avant tout à des utilisateurs expérimentés d'OCaml. En d'autres termes, si vous ne connaissez pas OCaml, il n'est peut-être pas judicieux de commencer la lecture ici.

Comme de coutume, je vous invite à me faire part de vos suggestions et critiques à propos de ce tutoriel. J'adresse en particulier ce message aux programmeurs C expérimentés qui pourront probablement corriger certaines fonctions maladroites.

Dernière chose, chaque partie du tutoriel se termine par un tableau récapitulatif des macros et fonctions qui y ont été abordées (ceci pour faciliter la recherche des notions dans le texte).

Bonne lecture,
Cacophrène
Sommaire du tutoriel :

- Des macros et de la discipline
- Hello world!
- Les types de base
- Au pays des allocations
- Vivre en harmonie avec le ramasse-miettes
- N-uplets et listes
- Où l'on déshabille un chameau
- Erreurs fréquentes

Des macros et de la discipline

Des macros

Une **macro** est un fragment de code auquel a été donné un nom. Un programme, le **préprocesseur**, remplace les macros par leur contenu dans le code source avant de le donner au compilateur. Notons au passage que le préprocesseur du langage C est très différent du préprocesseur camlp4 : alors que le premier effectue essentiellement des substitutions de texte, le second agit sur un arbre syntaxique abstrait (AST).

Pour que le dialogue entre OCaml et le C soit facile à mettre en œuvre et sûr, les développeurs d'OCaml ont défini de nombreuses macros. Elles sont toutes définies dans les fichiers d'en-tête du sous-répertoire caml de votre installation :

Code : Console

```
$ ls $(ocamlc -where)/caml
alloc.h     compatibility.h  fail.h    misc.h
bigarray.h  config.h         intext.h  mlvalues.h
callback.h  custom.h         memory.h  printexc.h
```

Nous n'allons pas utiliser tous ces fichiers dans ce tutoriel, c'est pourquoi je ne vous donne la description que de quelques-uns d'entre eux :

Fichier d'en-tête	Contenu
mlvalues.h	Macros et fonctions usuelles, type value
fail.h	Lever des exceptions
alloc.h	Allouer de la mémoire
memory.h	Dialoguer avec le GC

De la discipline

Tous les types OCaml sont exportés dans le monde C avec le type unique value. On convertit ensuite les valeurs de ce type en données manipulables par le C, et on renvoie une valeur qui, elle aussi, doit être de type value.

Le code C et le code OCaml sont placés dans des fichiers séparés qui diffèrent par leur nom. En effet, si vous choisissez foo.c et foo.ml, les fichiers ne diffèrent que par leur extension... or, à la compilation, vous allez créer deux fichiers foo.o : un pour OCaml et un pour le C. Vous l'aurez compris, cela ne marchera pas. Sachez également que la coutume veut que l'on utilise les noms foo_stub.c et foo.ml. Je vais donc garder cette convention tout au long de ce tutoriel.

Hello world!

Pour débuter, faisons simple. Nous allons tenter d'écrire une fonction hello en C que nous appellerons depuis OCaml. Voici le code, que nous détaillerons après :

Code : C - hello_stub.c

```c
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <stdio.h>

CAMLprim
value caml_hello(value unit) {
  CAMLparam1(unit);
  printf("Hello world!\n");
  CAMLreturn(Val_unit);
}
```

Code : OCaml - hello.ml

```
external hello : unit -> unit = "caml_hello"

let _ = hello ()
```

Du côté d'OCaml, les choses sont assez simples. Les fonctions écrites en C (fonctions externes) sont définies avec le mot-clef external. Elles sont suivies de leur signature, puis d'une chaîne de caractères qui n'est autre que le nom correspondant dans le

code C. En résumé :

external nom_ocaml : signature = "nom_c"

La macro CAMLprim

Les choses se compliquent un peu du côté du code C. On trouve d'abord la macro CAMLprim, qui doit *toujours* précéder les fonctions C accessibles depuis OCaml. On trouve ensuite une définition assez classique de fonction en C, avec cette particularité que les arguments reçus en entrée ont tous le même type value, quel que soit leur type d'origine dans le monde OCaml.

La macro CAMLparamN

Le corps de la fonction diffère également d'un code C standard. Les arguments reçus en entrée sont tous protégés avec la macro CAMLparam*N* (*N* est à remplacer par le nombre d'arguments) pour s'assurer qu'aucune interaction malheureuse avec le ramasse-miettes (GC) d'OCaml ne viendra perturber le bon déroulement du programme.

La macro CAMLreturn

Dernière chose, et non des moindres : on doit renvoyer vers OCaml une valeur de type value en utilisant la macro CAMLreturn en lieu et place du mot-clef `return` habituel. Ce premier code me permet d'ailleurs d'introduire la macro Val_unit, qui est l'équivalent C de la valeur () (type unit).

Compilation

C'est tout ! Il ne reste plus qu'à compiler. Dans tous les exemples que nous allons présenter ici, c'est une étape facile puisqu'il suffit de passer les deux fichiers au compilateur OCaml (si vous voulez savoir ce qui se passe, utilisez l'option **-verbose**) :

ocamlopt hello_stub.c hello.ml -o hello

et ça marche ! (enfin ça devrait)

Commande/Macro	Fonction
CAMLprim	Introduit une fonction C accessible depuis OCaml
CAMLparam*N*	Protéger les *N* arguments reçus en entrée
CAMLreturn	Renvoyer un résultat (de type value) dans le monde OCaml
Val_unit	value (unit OCaml)

Les types de base

Voyons maintenant comment manipuler les types de base, c'est-à-dire les *entiers* et les *booléens*. Vous l'aurez compris, par « types de base », il faut en fait entendre les types qui ne font pas l'objet d'une allocation.

Les booléens

Commençons par les booléens. Par exemple, nous pouvons essayer de réécrire les tests logiques et (&&) et ou (||).

Code : C - bool_stub.c

```
#include <caml/mlvalues.h>
#include <caml/memory.h>

CAMLprim
value caml_and(value x, value y) {
  CAMLparam2(x, y);
  int res = Bool_val(x) && Bool_val(y);
  CAMLreturn(Val_bool(res));
}
```

```
CAMLprim
value caml_or(value x, value y) {
  CAMLparam2(x, y);
  int res = Bool_val(x) || Bool_val(y);
  CAMLreturn(Val_bool(res));
}
```

Code : OCaml - bool.ml

```
external my_and : bool -> bool -> bool = "caml_and"
external my_or : bool -> bool -> bool = "caml_or"

let _ =
  let print = Printf.printf "Résultat : %B\n%!" in
  print (my_and true false);
  print (my_or false true)
```

Ce code permet d'introduire deux nouvelles macros. La première, Bool_val, renvoie un entier C (0 ou 1) à partir d'une valeur de type value qui correspond à un booléen en OCaml. La seconde, Val_bool, assure la fonction inverse : elle renvoie une valeur de type value (correspondant au type bool d'OCaml) à partir d'un entier C.

Toutes les macros qui commencent par Val_ renvoient une valeur de type value. Toutes les autres convertissent une valeur de type value en quelque chose de compréhensible par le C. On peut appeler cette règle la règle *To_from*.

Bon, ça suffit !  Testons sans tarder notre code :

ocamlopt bool_stub.c bool.ml -o bool

Les entiers

Il existe des macros Int_val et Val_int, qui étendent aux entiers ce que nous venons de voir avec les booléens. Comme vous pouvez le voir, elles obéissent aussi à la règle générale de type *To_from* que nous avons exposée précédemment.

Code : C - int_stub.c

```
#include <caml/mlvalues.h>
#include <caml/memory.h>

CAMLprim
value caml_succ(value x) {
  CAMLparam1(x);
  int res = Int_val(x) + 1;
  CAMLreturn(Val_int(res));
}

CAMLprim
value caml_prev(value x) {
  CAMLparam1(x);
  int res = Int_val(x) - 1;
  CAMLreturn(Val_int(res));
}
```

Code : OCaml - int.ml

```
external succ : int -> int = "caml_succ"
```

```
external prev : int -> int = "caml_prev"

let _ =
  let n = 25 in
  Printf.printf "Résultat : %d < %d < %d\n%!" (pred n) n (succ n)
```

Ce code peut être compilé avec **ocamlopt int_stub.c int.ml -o int**.

Commande/Macro	Fonction
Int_val	value vers entier C
Val_int	Entier C vers value (entier OCaml)
Bool_val	value vers entier C (0 ou 1)
Val_bool	Entier C vers value (booléen OCaml)
Val_true	value (booléen OCaml true)
Val_false	value (booléen OCaml false)

Au pays des allocations

Maintenant que nous avons vu les cas les plus simples, nous pouvons aborder le cas des *nombres à virgule flottante* et des *chaînes de caractères*. Ces types diffèrent des précédents dans la mesure où les valeurs correspondantes font l'objet d'une allocation.

Les nombres à virgule flottante

Je vous propose d'écrire des fonctions de calcul de l'*exponentielle naturelle* et du *logarithme népérien* d'un flottant :

Code : C - math_stub.c

```
#include <math.h>
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/alloc.h>

CAMLprim
value caml_exp(value x) {
  CAMLparam1(x);
  double res = exp(Double_val(x));
  CAMLreturn(caml_copy_double(res));
}

CAMLprim
value caml_exp(value x) {
  CAMLparam1(x);
  double res = log(Double_val(x));
  CAMLreturn(caml_copy_double(res));
}
```

Code : OCaml - math.ml

```
external my_exp : float -> float = "caml_exp"
external my_log : float -> float = "caml_log"

let   =
```

```
  let resultat = my_exp 1.0 in
  Printf.printf "Résultat : %.4f %.4f\n%!" (log resultat) (my_log
resultat)
```

Macros d'allocation

Notez l'utilisation du fichier alloc.h dans lequel sont définies les fonctions d'allocation. Parmi les nouveautés, on trouve la fonction caml_copy_double qui permet de convertir un flottant C (qu'il soit de type float ou double) en valeur de type value, et Double_val, qui renvoie un flottant C de type double à partir d'une valeur de type value.

Quelques remarques

Remarque 1 : le type float d'OCaml correspond au type double du C. En d'autres termes, OCaml n'a pas de nombres à virgule flottante en précision simple.
Remarque 2 : en C comme en OCaml, le logarithme népérien est noté log et le logarithme décimal log10. Il n'y a donc pas de confusion !

Comme précédemment, on compile avec :

ocamlopt math_stub.c math.ml -o math

Les chaînes de caractères

Qu'en est-il des chaînes de caractères ? Eh bien, c'est un peu la même chose. Il existe une macro String_val qui renvoie un char* à partir d'une valeur de type value, et une fonction caml_copy_string pour l'opération inverse. Voyez par exemple :

Code : C - str_stub.c

```
#include <ctype.h>
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/alloc.h>

CAMLprim
value caml_uppercase(value s) {
  CAMLparam1(s);
  int i;
  char *str = String_val(s);
  for(i = 0; i < caml_string_length(s); i++)
    str[i] = toupper(str[i]);
  CAMLreturn(s);
}
```

Code : OCaml - str.ml

```
external uppercase : string -> string = "caml_uppercase"

let _ = print_endline (uppercase "Hello world!")
```

Ce code peut être compilé comme précédemment :

ocamlopt str_stub.c str.ml -o str

Les chaînes de caractères OCaml peuvent contenir le caractère null (\000). Il est donc préférable d'utiliser caml_string_length au lieu de strlen si l'on ne sait pas exactement quel sera le contenu de la chaîne à traiter.

Commande/Macro	Fonction
Double_val	value vers flottant C
caml_copy_double	Flottant C vers value (flottant OCaml)
String_val	value vers char*
caml_string_length	Renvoie la longueur d'une chaîne OCaml (type value) sous forme d'entier C
caml_copy_string	char* C vers value (chaîne OCaml)

Vivre en harmonie avec le ramasse-miettes

Les exemples que nous venons de voir jusqu'à présent sont en fait très simples et masquent le problème principal posé par les allocations. Quel est-il ?

Vous savez sans doute qu'OCaml possède un *ramasse-miettes* (GC). Or, si vous souhaitez définir une valeur de type value à l'intérieur d'une fonction C (valeur locale), mais que vous voulez continuer à utiliser les valeurs reçues en argument, sachez qu'il existe un risque qu'elles soient récupérées par le GC au moment de l'allocation. Aïe !

Interactions avec le ramasse-miettes

Il faut donc préciser au GC que les valeurs reçues par la fonction C doivent être conservées. C'est pourquoi on utilise depuis le débutla macro CAMLparam*N* (en remplaçant *N* par le nombre de paramètres). C'est aussi pour cette raison que l'on utilise CAMLreturn à la place du mot-clef return.

Variables locales de type value

De la même façon, la définition de *variables locales* de type value se fera grâce à la macro CAMLlocal*N* (en remplaçant *N* par le nombre de paramètres). Pour mémoire, cette macro, comme les précédentes, nécessite le fichier d'en-tête memory.h.

Subtilités

Si vous décidez d'approfondir votre connaissance du dialogue entre le C et OCaml, vous apprendrez qu'il existe des cas où l'on peut se passer des macros CAMLparam*N* et CAMLreturn. Mais attention : **il est fortement recommandé de les utiliser systématiquement quand on débute**, comme nous le faisons dans ce tutoriel. En effet, il vaut mieux les utiliser dans des situations où elles sont superflues (y perd-on vraiment grand-chose ?) que les oublier là où elles sont utiles !

Que j'aime à faire apprendre ce nombre aux sages...

Pour illustrer l'utilisation de la macro CAMLlocal*N*, écrivons une fonction C qui renvoie une valeur approchée de π en utilisant la formule $\pi = \arccos(-1)$.

Code : C - pi_stub.c

```
#include <math.h>
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/alloc.h>

CAMLprim
value caml_approx_pi(value unit) {
  CAMLparam1(unit);
  CAMLlocal1(approx_pi);
  approx_pi = caml_copy_double(acos(-1));
  CAMLreturn(approx_pi);
}
```

Code : OCaml - pi.ml

```
external approx_pi : unit -> float = "caml_approx_pi"

let _ = Printf.printf "pi = %.12f\n%!" (approx_pi ())
```

On compile comme d'habitude (vous devez avoir pris l'habitude depuis que je l'écris) :

ocamlopt pi_stub.c pi.ml -o pi

Commande/Macro	Fonction
CAMLparam*N*	Préserve les valeurs reçues en entrée
CAMLlocal*N*	Définit des valeurs locales de type value
CAMLreturn	Renvoie une valeur de type value

N-uplets et listes

Nous allons maintenant nous intéresser au parcours et à la construction de types plus complexes tels que les n-uplets et les listes.

Les n-uplets

Inspection

Les n-uplets (tuples en anglais) sont constitués d'un nombre variable de champs de types hétérogènes. On accède à un champ donné avec la commande Field(tuple, i) où tuple est le n-uplet (de type value) et i l'index du champ auquel on veut accéder. Les champs sont numérotés à partir de zéro. La valeur renvoyée par Field est elle-même de type value.

Nous pouvons donc écrire une fonction très générale qui reçoit en entrée un triplet et un entier et renvoie le champ correspondant. Lorsque l'entier reçu en argument est incorrect, l'exception **`Invalid_argument`** est levée (nous n'avons pas parlé des exceptions dans ce tutoriel, mais leur utilisation dans le cas présent est assez intuitive) :

Code : C - tuple_stub.c

```
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/fail.h>

CAMLprim
value caml_triplet_nth(value triplet, value n) {
  CAMLparam2(triplet, n);
  int i = Int_val(n);
  if (i < 0 || i > 2) caml_invalid_argument("triplet_nth");
  CAMLreturn(Field(triplet, i));
}
```

Code : OCaml - tuple.ml

```
external triplet_nth : float * float * float -> int -> float =
"caml_triplet_nth"

let _ =
  let x = (1.6, 3.2, 7.5) in
  Printf.printf "x = (%.1f, %.1f, %.1f)\n%!" (triplet_nth x 0)
(triplet nth x 1)
```

```
        (triplet_nth x 2)
```

Cet exemple se compile comme tous les autres :

ocamlopt tuple_stub.c tuple.ml -o tuple

Création

La création d'un n-uplet fait appel à la commande caml_alloc_tuple(n) où *n* désigne le nombre de champs du n-uplet. Des valeurs peuvent ensuite être stockées dans les champs à l'aide de la commande Store_field(tuple, i, value). Voici un exemple :

Code : C - mktuple_stub.c

```
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/alloc.h>

CAMLprim
value caml_triplet(value unit) {
  CAMLparam1(unit);
  CAMLlocal1(triplet);
  triplet = caml_alloc_tuple(3);
  Store_field(triplet, 0, Val_true);
  Store_field(triplet, 1, caml_copy_double(3.14));
  Store_field(triplet, 2, Val_int(65));
  CAMLreturn(triplet);
}
```

Code : OCaml - mktuple.ml

```
external triplet : unit -> bool * float * char = "caml_triplet"

let _ =
  let x, y, z = triplet () in
  Printf.printf "(%b, %.2f, %C)\n%!" x y z
```

Je vous laisse le soin de compiler comme des grands. 

Les listes

Parcours

Les listes d'OCaml sont représentées à l'aide de couples composés d'une tête h (un élément de la liste) et d'une queue t (la sous-liste restante). On accède au contenu de ces couples avec Field, comme précédemment.

La *liste vide* fait l'objet d'un traitement particulier. Elle est obtenue avec la macro Val_emptylist qui est un synonyme de Val_int(0).

Code : C - list_stub.c

```
#include <stdio.h>
#include <caml/mlvalues.h>
#include <caml/memory.h>
```

```
CAMLprim
value caml_inspect_list(value list) {
  CAMLparam1(list);
  CAMLlocal1(head);
  while (list != Val_emptylist) {
    head = Field(list, 0);
    printf("%s\n", String_val(head));
    list = Field(list, 1);
  }
  CAMLreturn(Val_unit);
}
```

Code : OCaml - list.ml

```
external inspect_list : string list -> unit = "caml_inspect_list"

let _ = inspect_list ["Hello"; "world"; "!"]
```

Compilez avec **ocamlopt list_stub.c list.ml -o list**.

Construction

Pour construire une liste, on procède exactement de la même façon. On crée un couple pour chaque élément de la liste. Le premier élément du couple contient la valeur (de type value) et le second la queue de la liste. Pour créer un couple, on utilise la commande caml_alloc(n, tag). Le paramètre *n* indique la taille du bloc à allouer (2 pour le couple tête/queue). Le paramètre tag vaut toujours 0 dans le cas d'une liste. Il s'agit d'une valeur (étiquette) qui **renseigne sur la nature des données** (par exemple, il existe un tag pour les fermetures, les objets, les chaînes de caractères, etc.). On utilise ensuite Store_field(list, 0, x) pour stocker la tête de liste (ici *x*) et Store_field(list, 1, y) pour stocker la queue (ici *y*).

Code : C - explode_stub.c

```
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/alloc.h>

CAMLprim
value caml_explode(value s) {
  CAMLparam1(s);
  CAMLlocal2(list, cons);
  list = Val_emptylist;
  int i;
  char* str = String_val(s);
  for(i = caml_string_length(s); i; i--) {
    cons = caml_alloc(2, 0);
    Store_field(cons, 0, Val_int(str[i - 1]));
    Store_field(cons, 1, list);
    list = cons;
  }
  CAMLreturn(list);
}
```

Code : OCaml - explode.ml

```
external explode : string -> char list = "caml_explode"

let _ =
  let str = read_line () in
  Printf.printf "[ ";
```

```
        List.iter (Printf.printf "%C ") (explode str);
        Printf.printf "]\n%!"
```

Compilez avec **ocamlopt explode_stub.c explode.ml -o explode**.

Commande/Macro	Fonction
Field(v, n)	Renvoie le champ *n* (entier C) de la valeur *v* (type value)
Store_field(v, n, x)	Enregistre la valeur *x* (type value) dans le champ *n* (entier C) de la valeur *v* (type value)
caml_alloc(n, tag)	Alloue un bloc de taille *n* (entier C) contenant l'étiquette *tag* (entier C)
caml_alloc_tuple(i)	Alloue de la mémoire pour un n-uplet contenant *i* champs
caml_alloc_string(n)	Alloue de la mémoire pour une chaîne de caractères de longueur *n*. La chaîne est initialisée avec des données quelconques.

Où l'on déshabille un chameau

Des avantages et des inconvénients

Terminons cette présentation du dialogue entre C et OCaml par quelques considérations générales sur ce que nous venons de voir. D'abord, j'espère vous avoir convaincu que l'ajout de code C dans un code OCaml *n'est pas* une chose anodine : il y a des règles à respecter et des risques potentiels à prendre en considération.

OCaml vu de l'intérieur

Cette présentation nous a également permis de mettre à jour une partie de la *représentation interne* des types d'OCaml. C'est cette même représentation interne qu'il vous est possible de manipuler avec le module de magie noire Obj.

Le fantasme de la fonction print polymorphe

Je crois que vous êtes maintenant en mesure de comprendre pourquoi il n'est pas possible d'écrire une fonction print polymorphe en OCaml. Puisque les informations de type sont perdues à l'exécution, on ne peut afficher que la représentation interne d'une valeur. Ceci a pour conséquence que la fonction print polymorphe affichera la même chose pour plusieurs entrées différentes, comme (), [], None, 0 et false.

Erreurs fréquentes

Dans cette partie, nous allons voir quelles sont les principales erreurs auxquelles vous pouvez être confronté lorsque vous écrivez des fonctions de dialogue entre OCaml et le C.

Soyons clair : Cette partie du tutoriel contient des codes erronés susceptibles de planter plus ou moins méchamment (du résultat incorrect à l'erreur de segmentation).

Des calculs erronés

Voyons d'abord ce qui se passe quand on oublie de convertir un résultat de calcul en value. Pour cela, considérons la fonction ***erronée*** suivante :

Code : C - int_stub.c

```
#include <caml/mlvalues.h>
#include <caml/memory.h>

CAMLprim
value caml_succ(value n) {
  CAMLparam1(n);
```

```
  int res = Int_val(n) + 1;
CAMLreturn(res);
}
```

Code : OCaml - int.ml

```
external succ : int -> int = "caml_succ"

let _ =
  let n = 3 in
  Printf.printf "succ %d = %d\n%!" n (succ n)
```

À l'exécution, on obtient quelque chose d'assez inattendu :

Code : Console

```
$ ocamlopt int_stub.c int.ml -o int
$ ./int
succ 4 = 2
$
```

Si vous avez bien suivi le tutoriel, vous avez dû remarquer que l'on a oublié de convertir la variable res en valeur de type value avant de la renvoyer à OCaml (regardez la ligne surlignée dans le code C). Sans plus attendre, compilons ce code (**ocamlopt int_stub.c int.ml -o int**) et exécutons-le : nous obtenons le texte "succ 3 = 2". Voilà qui est surprenant !

Pour bien comprendre ce qui ne va pas, il faut se souvenir que les entiers d'OCaml sont codés sur 31 bits (63 bits sur les architectures en 64 bits). Le dernier bit, c'est-à-dire le bit de poids faible en représentation binaire conventionnelle, est utilisé par le ramasse-miettes : il permet de savoir si l'on a affaire à un bloc (bit égal à 0) ou à un nombre (bit égal à 1).

Que se passe-t-il si l'on renvoie res sans conversion ? La variable res contient la valeur 4, qui se note **0b10**0 en binaire. Pour OCaml, ce résultat est valide mais correspond au chiffre 2 (partie en gras). De la même façon, si vous demandez le successeur de 4, res contiendra **0b10**1, qui correspond toujours à 2 pour OCaml !

Mise en échec du typage

Nous allons maitenant voir comment obtenir une erreur de segmentation (segfault). J'ai volontairement choisi un code *correct* pour montrer que l'erreur ne vient pas toujours d'une mauvaise programmation.

Code : C - crash_stub.c

```
#include <caml/mlvalues.h>
#include <caml/memory.h>

CAMLprim
value caml_nth_tuple(value tuple, value n) {
  CAMLparam2(tuple, n);
  CAMLreturn(Field(tuple, Int_val(n) - 1));
}
```

Code : OCaml - crash.ml

```
external nth_tuple : 'a -> int -> 'b = "caml_nth_tuple"

let   =
```

```
  let tuple = (1, "oui", true, None) in
Printf.printf "Résultat : %s\n%!" (nth_tuple tuple 3)
```

Voici ce que l'on obtient à l'exécution (le message d'erreur exact peut varier selon le système et la configuration locale) :

Code : Console

```
$ ocamlopt crash_stub.c crash.ml -o crash
$ ./crash
Erreur de segmentation
$
```

La ligne surlignée devrait vous mettre sur la piste : l'appel de `nth_tuple tuple 3` renvoie un **booléen** alors que la fonction `Printf.printf` attend une **chaîne de caractères**. Pourquoi l'erreur de typage n'est-elle pas signalée à la compilation comme c'est toujours le cas en OCaml ? Eh bien, la réponse est simple : la fonction **nth_tuple** est polymorphe; il est donc impossible de voir le problème.

Autres erreurs possibles

Hélas, la liste est longue et ne s'arrête sûrement pas là ! Il y a plein d'autres manières de provoquer une erreur de segmentation sans le vouloir. Comme il n'est pas possible de présenter tous les cas, je vous invite à suivre une règle simple : **testez vos codes le plus souvent possible**, bien avant d'avoir écrit plusieurs pages. Vous gagnerez un temps considérable !
Vous savez désormais comment fonctionne le dialogue entre OCaml et le C, au moins dans ses grandes lignes. Je crois que c'est suffisant pour vous permettre de faire appel au C dans des situations simples. Mais, n'en doutez pas, l'histoire n'est pas terminée, loin s'en faut !

Je vous parlerai dans un autre tutoriel des tableaux, des variants polymorphes, des types personnalisés, et peut-être aussi du passage en argument de fonctions OCaml (fermetures) à un code C. J'ai choisi de passer ces notions sous silence pour le moment car j'en ai déjà bien assez dit pour une introduction !

Bonne programmation et à bientôt,
Cacophrène

Remerciements

Je tiens à remercier **bluestorm** pour sa relecture critique et **Thunderseb** pour l'intérêt qu'il a porté à la validation de ce tutoriel.

Références bibliographiques

- Le dialogue entre OCaml et le C (complet)
- OCaml - Manuel de l'utilisateur
- Compilation C/OCaml et C++/OCaml

Partager